## Ключевые концепции React

1. Что такое React?

React это открытая JavaScript-библиотека для фронтенда, предназначенная для создания пользовательских интерфейсов, особенно, если речь идет о создании одностраничного приложения. Она отвечает за слой представления (view layout) в веб и мобильных приложениях. React был создан [Jordan Walke](https://github.com/jordwalke), разработчиком программного обеспечения из Facebook. React был представлен на Facebook News Feed в 2011 году, а для Instagram - в 2012 году.

1. Назовите основные особенности React

Основными особенностями React является следующее:

* 1. Использование VirtualDOM (виртуальной объектной модели документа) вместо RealDOM (настоящий или реальный DOM), поскольку манипуляции с RealDOM являются дорогостоящими с точки зрения производительности
  2. Поддержка рендеринга на стороне сервера (Server Side Rendering, SSR)
  3. Следование принципу однонаправленного потока или связывания данных
  4. Использование переиспользуемых/компонуемых компонентов пользовательского интерфейса (User Interface, UI) для формирования слоя представления

1. Что такое JSX?

JSX (JavaScript и XML) - это XML-подобный синтаксис, расширяющий возможности ECMAScript. По сути, он является синтаксическим сахаром для функции React.createElement(), совмещая выразительность JavaScript с HTML-подобным синтаксисом разметки.

В приведенном ниже примере, текст внутри тега h1 в методе render() возвращается в виде JavaScript-функции:

class App extends React.Component {

render() {

return (

<div>

<h1>{'Добро пожаловать в мир React!'}</h1>

</div>

)

}

}

### В чем разница между элементом и компонентом?

Элемент - это обычный объект, описывающий, что мы хотим увидеть на экране в терминах узлов и других частей DOM. Элементы могут содержать другие элементы в своих свойствах. Создавать элементы в React легко. После создания, элемент не подлежит изменению.

Объектное представление React-элемента выглядит так:

const element = React.createElement(

'div',

{id: 'login-btn'},

'Войти'

)

Функция React.createElement() возвращает такой объект:

{

type: 'div',

props: {

children: 'Войти',

id: 'login-btn'

}

}

Данный объект рендерится с помощью ReactDOM.render():

<div id='login-btn'>Login</div>

В отличие от элемента, компонент может определяться по-разному. Он может быть классом с методом render() (классовый компонент). Или же он может быть простой функцией (функциональный компонент). В любом случае, компонент принимает свойства (пропы, props от properties) на вход и возвращает JSX:

const Button = ({ onLogin }) =>

<div id={'login-btn'} onClick={onLogin}>Войти</div>

JSX транспилируется (преобразуется) в функцию React.createElement():

const Button = ({ onLogin }) => React.createElement(

'div',

{ id: 'login-btn', onClick: onLogin },

'Войти'

)

### Как в React создаются компоненты?

Существует два способа это сделать:

* 1. **Функциональные компоненты:** это простейший способ создания компонента. Эти функции являются "чистыми", принимают объект с пропами в качестве аргумента и возвращают элемент(ы):

function Greeting({ message }) {

return <h1>{`Привет, ${message}`}</h1>

}

* 1. **Классовые компоненты:** для определения компонента также можно использовать ES6-классы. Приведенный функциональный компонент может быть переписан следующим образом:

class Greeting extends React.Component {

render() {

return <h1>{`Привет, ${this.props.message}`}</h1>

}

}

### Когда лучше использовать классовый компонент, а когда функциональный?

Если компонент нуждается в состоянии или методах жизненного цикла, тогда используйте классовый компонент, иначе, используйте функциональный компонент.

**Обратите внимание:** в React 16.8 были представлены хуки, позволяющие использовать состояние, методы жизненного цикла и другие возможности классовых компонентов в функциях.

### Что такое "чистые" компоненты (Pure Components)?

React.PureComponent - это тоже самое, что React.Component, за исключением автоматической обработки метода shouldComponentUpdate(). При измнении пропов или состояния PureComponent автоматически выполнит их поверхностное сравнение. С другой стороны, Component такого сравнения по умолчанию не проводит. Поэтому компонент будет повторно рендерится до тех пор, пока не будет вызван метод shouldComponentUpdate().

### Что такое состояние (state) в React?

Состояние - это объект, содержащий некоторую информацию, которая может измениться в течение жизненного цикла компонента. Мы всегда должны стараться делать состояние настолько простым, насколько это возможно, и минимизировать количество компонентов без состояния.

Создадим компонент User с состоянием message:

class User extends React.Component {

constructor(props) {

super(props)

this.state = {

message: 'Добро пожаловать в мир React!'

}

}

render() {

return (

<div>

<h1>{this.state.message}</h1>

</div>

)

}

}

Состояние похоже на проп, но оно является приватным (замкнуто в своей области видимости) и полностью контролируется компонентом, т.е. оно недоступно для других компонентов, кроме того, которому оно принадлежит и которое его определяет.

1. Что такое пропы (props) в React?

Props - это входные данные для компонента. Это простые значения (примитивы) или объект, содержащий несколько значений, которые передаются компонентам при их создании с помощью синтаксиса, похожего на атрибуты HTML-тегов.

Основное назначение пропов в React заключается в предоставлении компоненту следующего функционала:

* 1. Передача данных компоненту
  2. Вызов изменения состояния
  3. Использование через this.props.reactProp внутри метода render() компонента

Создадим элемент со свойством reactProp:

<Element reactProp={'1'} />

Этот reactProp добавляется в качестве свойства ко встроенному объекту props, который присутствет во всех компонентах, созданных с помощью React.

props.reactProp

### В чем разница между состоянием и пропами?

И props, и state являются обычными JavaScript-объектами. Несмотря на то, что они оба содержат информацию, которая используется при рендеринге компонента, функционал у них разный. Пропы передаются компоненту подобно аргументам, передаваемым функции, а состояние управляется компонентом как переменные, объявленные внутри функции.

### Почему не следует обновлять состояние напрямую?

Если вы попытаетесь обновить состояние напрямую, компонент не будет повторно отрендерен:

// Неправильно

this.state.message = 'Привет, народ'

Вместо этого, следует использовать метод setState(). Он планирует (откладывает) обновление состояния компонента. Когда состояние меняется, компонент перерисовывается:

// Правильно

this.setState({ message: 'Привет, народ' })

**Обратите внимание:** состояние компонента можно изменять напрямую в constructor() или с помощью нового синтаксиса определения полей классов (данное предложение находится на 3 стадии рассмотрения).

### Для чего в setState() используются функции обратного вызова?

Коллбек вызывается после выполнения setState() и рендеринга компонента. Поскольку метод setState() является асинхронным, коллбек используется для выполнения любых "последующих" операций.

setState({ name: 'Иван' }, () => console.log(`Свойство "name" обновлено и компонент повторно отрисован`))

**Обратите внимание**: вместо таких коллбеков рекомендуется использовать методы жизненного цикла.

### В чем разница между обработкой событий в HTML и React?

Ниже приведены некоторые из основных отличий:

* 1. В HTML название события указывается строчными буквами (в нижнем регистре):

<button onclick='activateLasers()'>

В React для этого используется "верблюжий стиль" (camelCase):

<button onClick={activateLasers}>

* 1. В HTML можно вернуть false для предотвращения поведения по умолчанию:

<a href='#' onclick='console.log("Ссылка была нажата"); return false;' />

В React необходимо явно вызывать метод preventDefault():

function handleClick(event) {

event.preventDefault()

console.log('Ссылка была нажата')

}

* 1. В HTML необходимо вызывать функцию с помощью ()

В React этого делать не нужно.

### Как в коллбеках JSX привязываются методы или обработчики событий?

Существует 3 способа это сделать:

* 1. **Привязка в конструкторе:** в классах JavaScript методы не связаны с экземплярами по умолчанию. Тоже самое справедливо для обработчиков событий в React. Обычно, мы делаем привязку в конструкторе:

class Component extends React.Component {

constructor(props) {

super(props)

this.handleClick = this.handleClick.bind(this)

}

handleClick() {

// ...

}

}

* 1. **Синтаксис публичных полей класса:** если вам не нравится использовать bind(), для привязки коллбеков можно использовать синтаксис публичных полей класса:

handleClick = () => {

console.log('Это: ', this)

}

<button onClick={this.handleClick}>

Нажми на меня

</button>

* 1. **Стрелочные функции:** вы также можете использовать стрелочные функции прямо в коллбеках:

<button onClick={(event) => this.handleClick(event)}>

Нажми на меня

</button>

**Обратите внимание**: если коллбек передается в качестве пропа дочерним компонентам, это может привести к ненужному повторному рендерингу. В таких случаях, рекомендуется использовать bind() или синтаксис публичных полей класса для улучшения производительности.

1. Как передать аргумент в "коллбек" или обработчик событий?

Для этого можно использовать стрелочную функцию в качестве обертки для обработчика событий:

<button onClick={() => this.handleClick(id)} />

Это эквивалентно вызову bind():

<button onClick={this.handleClick.bind(this, id)} />

Кроме названных подходов, вы также можете передавать аргументы в стрелочную функцию:

<button onClick={this.handleClick(id)} />

handleClick = (id) => () => {

console.log("Здравствуйте, номер вашего билета: ", id)

};

1. Что такое синтетические события в React?

SyntheticEvent - это кроссбраузерная оболочка для нативных событий браузера. Этот API аналогичен браузерному, включая stopPropagation() и preventDefault(), но работает одинаково во всех браузерах.

1. Что такое условный рендеринг?

Для условного рендеринга можно использовать обычные if или тернарные операторы. Кроме того, вы можете встраивать в JSX любое выражение посредством оборачивания его в фигурные скобки, а также совместно с логическим оператором && (короткие вычисления).

<h1>Привет!</h1>

{

messages.length > 0 && !isLogin?

<h2>

У вас {messages.length} непрочитанных сообщений.

</h2>

:

<h2>

У вас нет непрочитанных сообщений.

</h2>

}

1. Что такое проп key и в чем заключаются преимущества его использования?

key - это специальный строковый атрибут, который вы должны использовать при создании списков элементов. Проп key помогает React определять, какие элементы подверглись изменениям, были добавлены или удалены.

Чаще всего, в качестве ключа мы используем идентификаторы:

const todoItems = todos.map((todo) =>

<li key={todo.id}>

{todo.text}

</li>

)

При отсутствии ID, в качестве ключа можно использовать индекс элемента:

const todoItems = todos.map((todo, index) =>

<li key={index}>

{todo.text}

</li>

)

Обратите внимание:

* 1. Использовать индексы в качестве ключей не рекомендуется, если порядок расположения элементов может измениться. Это может негативно сказаться на производительности, а также привести к проблемам с состоянием компонента
  2. При извлечении элемента списка в качестве самостоятельного компонента применяйте ключи к этим компонентам, а не к тегу li
  3. При отсутствии пропа key в консоль будет выведено соответствующее предупреждение

1. Для чего используются ссылки (рефы, refs)?

ref возвращает ссылку на DOM-элемент. Этого в большинстве случаев следует избегать. Тем не менее, ссылки могут быть полезны при необходимости получения прямого доступа к DOM-элементу или экземпляру компонента.

1. Как создаются рефы?

Существует два подхода:

* 1. Это новый подход. Ссылки создаются с помощью React.createRef() и привязываются к элементу через ref. Для того, чтобы иметь возможность использовать рефы во всем компоненте, просто присвойте ref свойству экземпляра в конструкторе:

class MyComponent extends React.Component {

constructor(props) {

super(props)

this.myRef = React.createRef()

}

render() {

return <div ref={this.myRef} />

}

}

* 1. Также можно использовать реф-коллбеки (callback refs). Например, доступ к полю для ввода текста поисковой строки можно получить следующим образом:

class SearchBar extends Component {

constructor(props) {

super(props);

this.txtSearch = null;

this.state = { term: '' };

this.setInputSearchRef = e => {

this.txtSearch = e;

}

}

onInputChange(event) {

this.setState({ term: this.txtSearch.value });

}

render() {

return (

<input

value={this.state.term}

onChange={this.onInputChange.bind(this)}

ref={this.setInputSearchRef} />

);

}

}

Вы также можете использовать рефы в функциональных компонентах с помощью замыканий или хука useRef().

Обратите внимание: использовать встроенные реф-коллбеки не рекомендуется.

1. Что такое передача ссылки (Forward Ref)?

Передача ссылки - это "фича" (feature - возможность, способность), которая позволяет компонентам принимать реф и передавать его потомкам:

const ButtonElement = React.forwardRef((props, ref) => (

<button ref={ref} className="CustomButton">

{props.children}

</button>

));

// Создаем ссылку на кнопку

const ref = React.createRef();

<ButtonElement ref={ref}>Передать ссылку</ButtonElement>

1. Что лучше использовать, "коллбек-рефы" или findDOMNode()?

Лучше использовать callback refs вместо API findDOMNode(). Это объясняется тем, что findDOMNode() препятствует будущим улучшениям React.

Типичный пример использования findDOMNode():

class MyComponent extends Component {

componentDidMount() {

findDOMNode(this).scrollIntoView()

}

render() {

return <div />

}

}

Рекомендуемый подход:

class MyComponent extends Component {

constructor(props){

super(props);

this.node = createRef();

}

componentDidMount() {

this.node.current.scrollIntoView();

}

render() {

return <div ref={this.node} />

}

}

1. Что такое виртуальный DOM?

Virtual DOM (VDOM) - это представление Real DOM, хранимое в оперативной памяти. Это представление синхронизируется с "настоящим" DOM. Сравнение происходит между вызовом функции рендеринга и отображением элемента на экране. Данный внутренний процесс называется reconciliation (согласованием).

1. Как работает виртуальный DOM?

Virtual DOM работает следующим образом:

* 1. При любом изменении внутренних данных пользовательский интерфейс меняется в представлении виртуального DOM.
  2. Затем вычисляется разница между предыдущим и новым представлениями.
  3. После этого обновляются только те части реального DOM, которые подверглись изменениям.

1. В чем разница между теневым (Shadow) и виртуальным DOM?

Shadow DOM - это браузерная технология, спроектированная для ограничения области видимости переменных и CSS в веб-компонентах. Virtual DOM - это концепция, реализуемая некоторыми библиотеками JavaScript поверх браузерных API.

1. Что такое React Fiber?

Fiber (волокно) - это новый движок согласования, изменение основного алгоритма в React 16. Основной задачей React Fiber является повышения производительности в таких областях, как анимация, создание макета страницы, обработка жестов, возможность приостанавливать, прерывать или повторно запускать выполнение операций, предоставление приоритета определенным типам обновлений, а также новые примитивы параллелизма.

1. Для чего предназначен React Fiber?

Цель React Fiber - повышение производительности в таких областях, как анимация, создание макета страницы и обработка жестов. Основной его особенностью является incremental rendering (инкрементальный рендеринг, используется Angular): возможность разделения процесса рендеринга на части и их объединение через различные фреймы.

1. Что такое управляемые компоненты?

Компоненты, которые управляют инпутами формы для текущего пользователя, называются управляемыми. Любое изменение состояния имеет соответствующий обработчик.

Например, для того, чтобы значение было представлено прописными буквами (в верхнем регистре), мы используем такой обработчик:

handleChange(event) {

this.setState({value: event.target.value.toUpperCase()})

}

1. Что такое неуправляемые компоненты?

Неуправляемые компоненты - это компоненты, которые хранят собственное состояние, при необходимости получить их текущее значение используются ссылки на DOM-элементы. Это больше похоже на обычный HTML.

В приведенном ниже примере мы получаем доступ к полю для ввода имени по ссылке:

class UserProfile extends React.Component {

constructor(props) {

super(props)

this.handleSubmit = this.handleSubmit.bind(this)

this.input = React.createRef()

}

handleSubmit(event) {

alert('Было отправлено имя: ' + this.input.current.value)

event.preventDefault()

}

render() {

return (

<form onSubmit={this.handleSubmit}>

<label>

Имя:

<input type="text" ref={this.input} />

</label>

<input type="submit" value="Submit" />

</form>

);

}

}

Для обработки форм рекомендуется использовать управляемые компоненты.

1. В чем разница между createElement() и cloneElement()?

JSX транспилируется в функции createElement() для создания элементов React, которые используются для объектного представления пользовательского интерфейса. А cloneElement() используется для клонирования элемента и передачи ему новых свойств.

1. Что такое поднятие состояния в React?

Когда несколько компонентов нуждаются в использовании одних и тех же изменяющихся данных, рекомендуется поднимать совместно используемое состояние до ближайшего общего предка. Это означает, что если два дочерних компонента используют одинаковые данные, следует поднять состояние к их родителю вместо дублирования состояния в каждом потомке.

1. Назовите стадии жизненного цикла компонента

Жизненный цикл компонента состоит из 3 стадий:

* 1. Монтирование: компонент готов к встраиванию в браузерный DOM. Эта стадия охватывает инициализацию в constructor(), а также методы жизненного цикла getDerivedStateFromProps(), render() и componentDidMount().
  2. Обновление: на данной стадии компонент обновляется либо из-за получения новых пропов, либо из-за обновления состояния с помощью setState() или forceUpdate(). Эта стадия охватывает такие методы жизненного цикла как getDerivedStateFromProps(), shouldComponentUpdate(), render(), getSnapshotBeforeUpdate() и componentDidUpdate().
  3. Размонтирование: на этой последней стадии компонент удаляется из браузерного DOM. Данная стадия включает метод жизненного цикла componentWillUnmount().

Следует отметить, что в React также имеются особые стадии применения изменений к DOM:

* 1. Рендеринг: компонент рендерится без каких-либо побочных эффектов. Это применяется в отношении "чистых" компонентов. На данной стадии React может приостанавливать, прерывать или перезапускать рендеринг.
  2. Pre-commit: перед обновлением компонента есть момент, когда React читает DOM через getSnapshotBeforeUpdate().
  3. Commit: React изменяет DOM и выполняет завершающие методы жизненного цикла, такие как componentDidMount() при монтировании, componentDidUpdate() при обновлении и componentWillUnmount() при размонтировании.

Стадии в React 16.3+ (или [интерактивная версия](http://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/))

1. Назовите методы жизненного цикла в React

До React 16.3:

* + componentWillMount: выполняется перед рендерингом для настройки корневого компонента на уровне приложения
  + componentDidMount: выполняется после первого рендеринга, здесь выполняются AJAX-запросы, обновляется DOM или состояние компонента, регистрируются обработчики событий
  + componentWillReceiveProps: выполняется при обновлении определенного пропа для запуска перехода состояния
  + shouldComponentUpdate: определяет, должен ли компонент обновляться. Значением по умолчанию является true. Если вы уверены в том, что компонент не нуждается в повторном рендеринге при изменении состояния или пропов, тогда можете вернуть ложное значение. Это подходящее место для улучшения производительности, позволяющее предотвратить ненужные рендеринги при получении компонентом новых пропов
  + componentWillUpdate: выполняется перед повторным рендерингом компонента при изменении состояния или пропов и при истинном значении, возвращаемом shouldComponentUpdate()
  + componentDidUpdate: в основном, используется для обновления DOM в соответствии с изменениями состояния или пропов
  + componentWillUnmount: используется для отмены сетевых запросов или удаления обработчиков событий, связанных с компонентом

React 16.3+:

* + getDerivedStateFromProps: запускается перед вызовом метода render() и при каждом повторном рендеринге. Он используется в редких случаях, когда нам требуется производное состояние. Для получения более подробной информации смотрите [если вам требуется производное состояние](https://reactjs.org/blog/2018/06/07/you-probably-dont-need-derived-state.html)
  + componentDidMount: выполняется после первого рендеринга, здесь выполняются AJAX-запросы, обновляется DOM или состояние компонента, регистрируются обработчики событий
  + shouldComponentUpdate: определяет, должен ли компонент обновляться. Значением по умолчанию является true. Если вы уверены в том, что компонент не нуждается в повторном рендеринге при изменении состояния или пропов, тогда можете вернуть ложное значение. Это подходящее место для улучшения производительности, позволяющее предотвратить ненужные рендеринги при получении компонентом новых пропов
  + getSnapshotBeforeUpdate: выполняется перед применением результатов рендеринга к DOM. Любое значение, возвращенное этим методом, передается в componentDidUpdate(). Это может быть полезным для получения информации из DOM, например, позиции курсора или величины прокрутки
  + componentDidUpdate: в основном, используется для обновления DOM в соответствии с изменением состояния или пропов. Не выполняется, если shouldComponentUpdate() возвращает false
  + componentWillUnmount используется для отмены сетевых запросов или удаления обработчиков событий, связанных с компонентом

1. Что такое компонент высшего порядка (Higher Order Component, HOC)?

Higher-order component (HOC) - это функция, принимающая компонент и возвращающая новый компонент. Это паттерн, производный от композиционной природы React.

Мы называем такие компоненты чистыми, поскольку они могут принимать и динамически предоставлять дочерние компоненты, но не меняют и не копируют их поведение.

const EnhancedComponent = higherOrderComponent(WrappedComponent)

HOC, обычно, используются для:

* 1. Обеспечения возможности переиспользования кода, логики, а также для абстрагирования шаблонов
  2. Отложенного рендеринга
  3. Абстрагирования и манипулирования состоянием
  4. Манипулирования пропами

1. Как в HOC-компоненте создаются прокси для пропов?

Вы можете добавлять/редактировать пропы, передаваемые в компонент, с помощью шаблона проксирования пропов следующим образом:

function HOC(WrappedComponent) {

return class Test extends Component {

render() {

const newProps = {

title: 'Новый заголовок',

footer: false,

showFeatureX: false,

showFeatureY: true

}

return <WrappedComponent {...this.props} {...newProps} />

}

}

}

1. Что такое контекст (Context)?

Context предоставляет возможность передавать данные в дереве компонента без необходимости передачи пропов на каждом уровне вручную.

Например, статус аутентификации пользователя, языковые предпочтения или цветовая схема могут использоваться многими компонентами приложения:

const { Provider, Consumer } = React.createContext(defaultValue)

1. Что такое проп children?

Потомки (дети) - это проп (this.props.children), позволяющий передавать одни компоненты другим, как любые другие пропы. Дерево компонентов, размещаемое между открывающим и закрывающим тегами, передается компоненту в качестве пропа children.

Для работы с этим пропом в React API существуют такие методы как: React.Children.map, React.Children.forEach, React.Children.count, React.Children.only и React.Children.toArray.

Простой пример использования пропа children:

const MyDiv = React.createClass({

render: function() {

return <div>{this.props.children}</div>

}

})

ReactDOM.render(

<MyDiv>

<span>Привет, </span>

<span>народ</span>

</MyDiv>,

node

)

1. Как выглядят комментарии в React?

Комментарии в React/JSX похожи на многострочные комментарии JavaScript, но оборачиваются в фигурные скобки:

Однострочные комментарии:

<div>

{/\* Однострочный комментарий (в ванильном JavaScript однострочные комментарии помещаются после двойного слеша (//)) \*/}

{`Добро пожаловать, ${user}. Давайте изучать React`}

</div>

Многострочные комментарии:

<div>

{/\* Комментарий, состоящий из

нескольких строк \*/}

{`Добро пожаловать, ${user}. Давайте изучать React`}

</div>

1. Для чего используется ключевое слово super с аргументом props в конструкторе?

Конструктор дочернего класса не может использовать ссылку на this до вызова метода super(). Тоже самое справедливо для ES6-подклассов. Основная причина передачи аргумента props в super() состоит в обеспечении возможности доступа к this.props в дочернем конструкторе.

Передача пропов:

class MyComponent extends React.Component {

constructor(props) {

super(props)

console.log(this.props) // вывод: { name: 'Иван', age: 30 }

}

}

Без пропов:

class MyComponent extends React.Component {

constructor(props) {

super()

console.log(this.props) // вывод: undefined

// однако, пропы по-прежнему доступны как в конструкторе, ...

console.log(props) // вывод: { name: 'Иван', age: 30 }

}

render() {

// так и за его пределами

console.log(this.props) // вывод { name: 'Иван', age: 30 }

}

}

1. Что такое согласование?

Когда изменяется состояние или пропы компонента, React определяет, нуждается ли DOM в обновлении посредством сравнения нового элемента с предыдущим. Когда эти элементы отличаются, React обновляет DOM. Данный процесс называется reconciliation.

1. Как определить состояние с помощью вычисляемого свойства?

При использовании ES6 или транспилятора Babel для преобразования JSX-кода вы можете применять вычисляемые названия свойств:

handleInputChange(event) {

this.setState({ [event.target.id]: event.target.value })

}

1. Какая распространенная ошибка приводит к вызову функции при каждом рендеринге?

При передачи функции в качестве аргумента необходимо убедиться, что она не вызывается:

render() {

// Неправильно: handleClick вызывается вместо передачи в качестве ссылки!

return <button onClick={this.handleClick()}>Нажми на меня</button>

}

Вместо этого, следует передавать саму функцию, без круглых скобок:

render() {

// Правильно: handleClick передается как ссылка!

return <button onClick={this.handleClick}>Нажми на меня</button>

}

1. Поддерживают ли "ленивые" (lazy) функции именованный экспорт?

Нет, функция React.lazy() поддерживает только экспорт по умолчанию. Если вы хотите импортировать модуль с помощью именованного импорта, то можете создать промежуточный модуль, который будет повторно экспортировать этот модуль по умолчанию. Это также позволит выполнять свою работу тряске дерева (tree shaking) и предотвратит загрузку неиспользуемых компонентов.

Рассмотрим файл компонента, в котором по имени импортируется несколько компонентов:

// MoreComponents.js

export const SomeComponent = /\* ... \*/;

export const UnusedComponent = /\* ... \*/;

Затем компоненты из MoreComponents.js повторно экспортируются в промежуточном файле IntermediateComponent.js:

// IntermediateComponent.js

export { SomeComponent as default } from "./MoreComponents.js";

После этого, компоненты могут импортироваться с помощью "ленивой" функции:

import React, { lazy } from 'react';

const SomeComponent = lazy(() => import("./IntermediateComponent.js"));

1. Почему в React используется className вместо атрибута class?

class - это одно из ключевых слов JavaScript, а JSX - это расширение JavaScript. Вот почему в React вместо class используется className. В качестве пропа className передается строка:

render() {

return <span className={'menu navigation-menu'}>Меню</span>

}

1. Что такое фрагмент (Fragment)?

Это распространенный паттерн в React, который используется в компонентах, возвращающих несколько элементов. Fragments позволяют группировать дочерние элементы без создания лишних DOM-узлов:

render() {

return (

<React.Fragment>

<ChildA />

<ChildB />

<ChildC />

</React.Fragment>

)

}

Также существует сокращенный синтаксис, но он не поддерживается в некоторых инструментах:

render() {

return (

<>

<ChildA />

<ChildB />

<ChildC />

</>

)

}

1. Почему фрагменты лучше, чем div?

Вот некоторые причины:

* 1. Фрагменты немного быстрее и используют меньше памяти. Реальная польза от этого ощущается в очень больших и глубоких деревьях элементов
  2. Некоторые механизмы CSS, например, Flexbox и Grid используют связь родитель-ребенок (предок-потомок, если угодно), поэтому добавление дополнительных div может сломать макет страницы.
  3. Удобнее пользоваться инспектором DOM

1. Что такое портал (Portal) в React?

Portal - рекомендуемый способ рендеринга потомков в DOM-узле, который находится за пределами родительского компонента:

ReactDOM.createPortal(child, container)

Первый аргумент - любой React-компонент, подлежащий рендерингу, такой как элемент, строка или фрагмент. Второй аргумент - DOM-элемент.

1. Что такое компонент без состояния?

Если поведение компонента не зависит от его состояния, такой компонент считается не имеющим состояния. Для создания компонентов без состояния можно использовать как функции, так и классы. Однако, если вам не нужны методы жизненного цикла, тогда лучше использовать функциональные компоненты. У функция по сравнению с классами в этом случае имеется несколько преимуществ: их легче писать, читать и тестировать, они немного быстрее и в них не требуется использовать ключевое слово this.

1. Что такое компонент с состоянием?

Если поведение компонента зависит от состояния, такой компонент считается имеющим состояние. Раньше компоненты с состоянием можно было создавать только с помощью классов. Сейчас хуки предоставляют возможность создавать функциональные компоненты с состоянием. В классах состояние инициализируется в constructor() с помощью this.state = initialValue, в функции, как правило, в начале с помощью const [state, setState] = useState(initialValue).

class App extends Component {

constructor(props) {

super(props)

this.state = { count: 0 }

}

render() {

// ...

}

}

React 16.8:

Эквивалентный функциональный компонент

import React, { useState } from 'react';

const App = (props) => {

const [count, setCount] = useState(0);

return (

// ...

)

}

1. Как осуществить проверку пропов в React?

При запуске приложения в режиме для разработки, React автоматически проверяет все пропы, определенные в компонентах, на правильные типы. Если тип является неправильным, React выводит в консоль предупреждение. Эта проверка отключена в режиме для продакшна с целью повышения производительности. Обязательные пропы определяются с помощью isRequired.

Набор предопределенных типов пропов:

* 1. PropTypes.number
  2. PropTypes.string
  3. PropTypes.array
  4. PropTypes.object
  5. PropTypes.func
  6. PropTypes.node
  7. PropTypes.element
  8. PropTypes.bool
  9. PropTypes.symbol
  10. PropTypes.any

Мы можем определить propTypes для компонента User следующим образом:

import React from 'react'

import PropTypes from 'prop-types'

class User extends React.Component {

static propTypes = {

name: PropTypes.string.isRequired,

age: PropTypes.number.isRequired

}

render() {

return (

<>

<h1>{`Добро пожаловать, ${this.props.name}`}</h1>

<h2>{`Возраст, ${this.props.age}`}</h2>

</>

)

}

}

Обратите внимание: В React 15.5 PropTypes были перемещены из React.PropTypes в библиотеку prop-types.

1. Какие преимущества предоставляет использование React?

Вот список основных преимуществ:

* 1. Повышение производительности приложения благодаря виртуальному DOM
  2. JSX облегчает написание и чтение кода
  3. Возможность рендеринга как на стороне клиента, так и на стороне сервера
  4. Возможность относительно простой интеграции с фреймворками (Angular, Backbone), поскольку React - это всего лишь библиотека
  5. Возможность быстрого юнит и интеграционного тестирования с помощью таких инструментов, как Jest

1. Какие ограничения имеются в React?

Кроме преимуществ, в React существуют некоторые ограничения:

* 1. React - это всего лишь библиотека, отвечающая за слой представления, а не полноценный фреймворк
  2. Его трудно изучать новичкам в веб-разработке
  3. Интеграция с традиционными MVC-фреймворками требует дополнительной настройки
  4. Код является более сложным из-за встроенных шаблонов и JSX
  5. Большое количество мелких компонентов приводит к сложности в проектировании и построении архитектуры приложения

1. Что такое предохранители (Error Boundaries) в React 16?

Error boundaries- это компоненты, которые отлавливают ошибки JavaScript, возникающие в любом дочернем компоненте, сообщают об этих ошибках и отображают резервный UI.

Классовый компонент становится предохранителем при определении в нем нового метода жизненного цикла componentDidCatch(error, info) или static getDerivedStateFromError() :

class ErrorBoundary extends React.Component {

constructor(props) {

super(props)

this.state = { hasError: false }

}

componentDidCatch(error, info) {

// Вы также можете отправлять ошибки в специальный сервис по их обработке

logErrorToMyService(error, info)

}

static getDerivedStateFromError(error) {

// Обновляем состояние, чтобы при следующем рендеринге использовался запасной UI

return { hasError: true };

}

render() {

if (this.state.hasError) {

// Вы можете рендерить любой резервный интерфейс

return <h1>{'Что-то пошло не так.'}</h1>

}

return this.props.children

}

}

После этого, предохранитель используется как обычный компонент:

<ErrorBoundary>

<MyWidget />

</ErrorBoundary>

1. Как реализовать предохранитель в React 15?

React 15 предоставляет поддержку для error boundaries с помощью метода unstable\_handleError(). Данный метод был переименован в componentDidCatch() в React 16.

1. Какой способ рекомендуется использовать для статической проверки типов?

Обычно, для проверки типов в React-приложениях используется библиотека PropTypes (React.PropTypes перемещен в пакет prop-types, начиная с React 15.5). Для больших приложений рекомендуется использовать статические типизаторы, такие как Flow или TypeScript, которые выполняют проверку типов во время компиляции и предоставляют возможности по автоматическому исправлению.

1. Для чего используется пакет react-dom?

react-dom предоставляет методы, специфичные для DOM, которые могут быть использованы на верхнем уровне приложения. Большинству компонентов не нужен этот модуль. Вот некоторые из методов рассматриваемой библиотеки:

* 1. render()
  2. hydrate()
  3. unmountComponentAtNode()
  4. findDOMNode()
  5. createPortal()

1. Для чего предназначен метод render() в react-dom?

Данный метод используется для встраивания React-элемента в DOM в определенный контейнер и возврата ссылки на этот элемент. Если React-элемент уже рендерился ранее, осуществляется обновление только той части DOM, которая подверглась изменениям.

ReactDOM.render(element, container[, callback])

При передаче опционального коллбека, он будет вызван после рендеринга или обновления.

1. Что такое ReactDOMServer?

ReactDOMServer - это объект, позволяющий рендерить компоненты в виде статической разметки (обычно, используется на node-серверах). Данный объект в основном используется при рендеринге на стороне сервера. Следующие методы могут быть использованы как на сервере, так и в браузерном окружении:

* 1. renderToString()
  2. renderToStaticMarkup()

Например, вы запускаете основанный на Node веб-сервер, такой как Express, Koa или Happi, и вызываете renderToString() для рендеринга корневого элемента в виде строки, которую сервер отправляет в ответ на запрос:

// используем Express

import { renderToString } from 'react-dom/server'

import MyPage from './MyPage'

app.get('/', (req, res) => {

res.write('<!DOCTYPE html><html><head><title>My Page</title></head><body>')

res.write('<div id="content">')

res.write(renderToString(<MyPage/>))

res.write('</div></body></html>')

res.end()

})

1. Как использовать InnerHtml в React?

Атрибут dangerouslySetInnerHTML в React является альтернативой innerHTML. Как и последний, его использование представляет собой угрозу межсайтового скриптинга (XSS). Необходимо передать объект с ключом \_\_html и HTML-разметкой в качестве значения.

В приведенном ниже примере MyComponent использует атрибут dangerouslySetInnerHTML для определения разметки:

function createMarkup() {

return { \_\_html: 'Первый &middot; Второй' }

}

function MyComponent() {

return <div dangerouslySetInnerHTML={createMarkup()} />

}

1. Как использовать стили в React?

Атрибут style принимает объект JavaScript со свойствами в стиле camelCase, а не в виде CSS-строки. Это обеспечивает согласованность с JavaScript-свойствами, связанными со стилями, является более эффективным и закрывает дыры в безопасности (XSS).

const divStyle = {

color: 'blue',

backgroundImage: 'url(' + imgUrl + ')'

};

function HelloWorldComponent() {

return <div style={divStyle}>Привет, народ!</div>

}

Стили пишутся в camelCase для интеграции со свойствами узлов DOM в JavaScript (например, node.style.backgroundImage).

1. Чем отличаются события в React?

Обработка событий в элементах React имеет некоторые синтаксические отличия:

* 1. Обработчики событий именуются в верблюжьем стиле, а не в нижнем регистре
  2. В JSX в обработчик передается функция, а не строка (функция передается как ссылка, т.е. не вызывается)

1. Что произойдет при использовании setState() в constructor()?

Если это сделать, то помимо присвоения значения объекту состояния React повторно отрендерит компонент и всех его потомков. Вы получите ошибку: Can only update a mounted or mounting component. (Можно обновлять только смонтированный или монтируемый компонент.) Поэтому для инициализации переменных внутри конструктора следует использовать this.state.

1. Почему следует избегать использования индексов в качестве ключей?

Ключи должны быть стабильными, предсказуемыми и уникальными, чтобы React имел возможность следить за элементами.

В приведенном ниже примере ключом каждого элемента является порядок его расположения в массиве без привязки к предоставляемым им данным. Это ограничивает возможности React по оптимизации:

{todos.map((todo, index) =>

<Todo

{...todo}

key={index}

/>

)}

При использовании в качестве уникальных ключей данных элемента, например, todo.id, которые являются уникальными в списке и стабильными, у React появится возможность изменять порядок расположения элементов без необходимости выполнения дополнительных вычислений:

{todos.map((todo) =>

<Todo {...todo}

key={todo.id} />

)}

1. Правильно ли использовать setState() в методе componentWillMount()?

Да, использование setState() внутри componentWillMount() является безопасным. В то же время, рекомендуется избегать выполнения асинхронной инициализации в методе жизненного цикла componentWillMount(). componentWillMount() вызывается перед монтированием. Он также вызывается перед методом render(), поэтому настройка состояния в этом методе не приводит к повторному рендерингу. В нем следует избегать любых побочных эффектов и подписок. Вместо componentWillMount() асинхронную логику следует помещать в componentDidMount().

componentDidMount() {

axios.get(`api/todos`)

.then((result) => {

this.setState({

messages: [...result.data]

})

})

}

1. Что произойдет при использовании пропов в начальном состоянии (Initial State)?

При изменении пропов компонента без его обновления, эти новые пропы никогда не будут отображены на экране, поскольку функция-конструктор никогда не обновит текущее состояние компонента. Обновление состояния через пропы возможно только после создания компонента.

В приведенном ниже примере обновленное значение инпута никогда не отобразится:

class MyComponent extends React.Component {

constructor(props) {

super(props)

this.state = {

records: [],

inputValue: this.props.inputValue

};

}

render() {

return <div>{this.state.inputValue}</div>

}

}

Значение обновится при использовании пропов в методе render():

class MyComponent extends React.Component {

constructor(props) {

super(props)

this.state = {

record: []

}

}

render() {

return <div>{this.props.inputValue}</div>

}

}

1. Как выполнить условный рендеринг компонентов?

В некоторых случаях вам нужно будет рендерить разные компоненты в зависимости от некоторого состояния. JSX не рендерит false или undefined, поэтому вы можете использовать короткие вычисления для рендеринга определенной части компонента только при удовлетворении условия:

const MyComponent = ({ name, address }) => (

<div>

<h2>{name}</h2>

{address &&

<p>{address}</p>

}

</div>

)

Если вам требуется условие if else, тогда используйте тернарный оператор:

const MyComponent = ({ name, address }) => (

<div>

<h2>{name}</h2>

{address

? <p>{address}</p>

: <p>{'Address is not available'}</p>

}

</div>

)

1. Почему следует быть осторожным при распространении (spread) пропов на DOM-элементы?

При распространении пропов мы подвержены риску добавления неизвестных HTML-атрибутов, что считается плохой практикой. Вместо этого, лучше использовать деструктуризацию пропа с помощью оператора ...rest. Это обеспечит добавление только нужных пропов:

Например:

const ComponentA = () =>

<ComponentB isDisplay={true} className='someClassName' />

const ComponentB = ({ isDisplay, ...domProps }) =>

<div {...domProps}>{ComponentB}</div>

1. Как использовать декораторы в React?

Вы можете декорировать ваши классовые компоненты способом, аналогичным передаче компонента в функцию. Decorators - гибкий и удобочитаемый способ модификации функциональности компонента:

@setTitle('Профиль')

class Profile extends React.Component {

//....

}

/\*

title - это строка, которая устанавливается заголовку документа

WrappedComponent - это то, что получит наш декоратор

при его размещении над классовым компонентом

\*/

const setTitle = (title) => (WrappedComponent) => {

return class extends React.Component {

componentDidMount() {

document.title = title

}

render() {

return <WrappedComponent {...this.props} />

}

}

}

Обратите внимание: декораторы - экспериментальная технология, находящаяся на 3 стадии рассмотрения.

1. Как запомнить или сохранить компонент?

Существуют библиотеки мемоизации, которые можно использовать в функциональных компонентах.

Например, библиотека moize может сохранять один компонент в другом:

import moize from 'moize'

import Component from './components/Component'

// этот модуль экспортирует немемоизированный компонент

const MemoizedFoo = moize.react(Component)

const Consumer = () => {

<div>

{/\* Это сохранит следующий компонент: \*/}

<MemoizedFoo />

</div>

}

Начиная с React 16.6.0, у нас имеется React.memo(). Это компонент высшего порядка, который сохраняет компонент неизменным до обновления его пропов. Для этого достаточно обернуть в него мемоизируемый компонент перед использованием последнего:

const MemoComponent = React.memo(function MemoComponent(props) {

/\* рендеринг с помощью пропов \*/

});

// или

export default React.memo(MyFunctionComponent);

Также следует отметить, что похожий функционал предоставляет хук useMemo().

1. Как реализовать рендеринг на стороне сервера или SSR?

React поддерживает рендеринг на стороне Node-сервера из коробки. Для этого используется специальная версия DOM-рендерера, которая реализует такой же паттерн, что и клиентская версия:

import ReactDOMServer from 'react-dom/server'

import App from './App'

ReactDOMServer.renderToString(<App />)

Этот метод возвращает обычный HTML в виде строки, которая затем может быть помещена в тело (body) ответа сервера. На стороне клиента React определяет предварительно отрендеренный контент и просто вставляет его в существующее дерево компонентов.

1. Как включить производственный режим в React?

Для этого следует установить NODE\_ENV в значение production в методе DefinePlugin() Webpack. Это отключит проверку типов и дополнительные предупреждения. Кроме того, при минификации кода, например, с помощью Uglify, удаляющего "мертвый" код, а также код и комментарии для разработки, размер сборки будет намного меньше.

1. Что такое CRA и в чем заключаются преимущества его использования?

create-react-app - это инструмент, позволяющий быстро создавать и запускать React-приложения, минуя стадию настройки.

Создадим Todo App с помощью CRA:

# Создание нового проект

$ npx create-react-app todo-app

// или

$ npm init react-app todo-app

// или

$ yarn create react-app todo-app

$ cd todo-app

# Сборка, тестирование и запуск

$ npm run build

$ npm run test

$ npm start

// или

$ yarn build

$ yarn test

$ yarn start

Проект включает в себя все необходимое для разработки приложения на React:

* 1. Поддержка синтаксиса ES6, React, JSX и Flow
  2. Расширения языка после ES6, такие как spread- и rest-операторы
  3. Автоматическое добавление префиксов к стилям, поэтому вам не нужно вручную добавлять -webkit- и другие префиксы
  4. Быстрый запуск интерактивных юнит-тестов со встроенной поддержкой отчетов о покрытии кода тестами
  5. Автоматически перезапускаемый сервер для разработки, выводящий предупреждения о наиболее распространенных ошибках
  6. Скрипт для сборки JS, CSS и статических файлов для продакшна с хэшами в названиях и картами ресурсов (sourcemaps)

1. Назовите методы жизненного цикла, относящиеся к монтированию

При создании и встраивании компонента в DOM методы жизненного цикла вызываются в следующем порядке:

* 1. constructor()
  2. static getDerivedStateFromProps()
  3. render()
  4. componentDidMount()

1. Какие методы жизненного цикла были признаны устаревшими в React 16?

Следующие методы жизненного цикла являются небезопасными практиками кодирования и усложняют работу с асинхронной логикой:

* 1. componentWillMount()
  2. componentWillReceiveProps()
  3. componentWillUpdate()

Начиная с React 16.3, эти методы следует использовать с префиксом UNSAFE\_, а версии без префиксов были удалены в React 17.

1. Для чего используется метод жизненного цикла getDerivedStateFromProps()?

Новый статический метод жизненного цикла getDerivedStateFromProps() вызывается после инстанцирования элемента перед его повторным рендерингом. Он может возвращать объект для обновления состояния или null как индикатор того, что новые пропы не требуют обновления состояния.

class MyComponent extends React.Component {

static getDerivedStateFromProps(props, state) {

// ...

}

}

Этот метод вместе с componentDidUpdate() охватывает все случаи использования componentWillReceiveProps().

1. Для чего используется метод жизненного цикла getSnapshotBeforeUpdate()?

Новый метод жизненного цикла getSnapshotBeforeUpdate() вызывается сразу после обновления DOM. Значение, возвращенное этим методом, передается в качестве третьего аргумента componentDidUpdate().

class MyComponent extends React.Component {

getSnapshotBeforeUpdate(prevProps, prevState) {

// ...

}

}

Этот метод вместе с componentDidUpdate() охватывает все случаи использования componentWillUpdate()

1. Заменяют ли хуки рендер-пропы и компоненты высшего порядка?

Рендер-пропы и компоненты высшего порядка рендерят только одного потомка, однако, в большинстве случаев, хуки предоставляют более простой способ минимизировать количество уровней вложенности дерева компонентов.

1. Как рекомендуется называть компоненты?

Компоненты рекомендуется именовать с помощью ссылок вместо использования displayName.

Использование displayName:

export default React.createClass({

displayName: 'TodoApp',

// ...

})

Рекомендуемый подход:

export default class TodoApp extends React.Component {

// ...

}

1. Какой порядок расположения методов в классовом компоненте является рекомендуемым?

Рекомендуемый порядок расположения методов от монтирования до рендеринга следующий:

* 1. static методы
  2. constructor()
  3. getChildContext()
  4. componentWillMount()
  5. componentDidMount()
  6. componentWillReceiveProps()
  7. shouldComponentUpdate()
  8. componentWillUpdate()
  9. componentDidUpdate()
  10. componentWillUnmount()
  11. обработчики событий, такие как handleSubmit() или handleChange()
  12. геттеры для рендеринга, такие как getSelectReason() или getFooterContent()
  13. опциональные методы рендеринга, такие как renderNavigation() или renderProfilePicture()
  14. render()

1. Что такое компонент-переключатель (Switching Component)?

Switching component - это компонент, который рендерит один из нескольких компонентов. Для получения значений пропов для компонентов необходимо использовать объект.

Например, вот компонент-переключатель, отображающий разные страницы на основе пропа page:

import HomePage from './HomePage'

import AboutPage from './AboutPage'

import ServicesPage from './ServicesPage'

import ContactPage from './ContactPage'

const PAGES = {

home: HomePage,

about: AboutPage,

services: ServicesPage,

contact: ContactPage

}

const Page = (props) => {

const Handler = PAGES[props.page] || ContactPage

return <Handler {...props} />

}

// ключи объекта PAGES могут быть использованы в propTypes для перехвата ошибок в процессе разработки

Page.propTypes = {

page: PropTypes.oneOf(Object.keys(PAGES)).isRequired

}

1. Для чего в setState() передается функция?

Дело в том, что setState() - это асинхронная операция. React откладывает обновление состояния по причинам производительности, поэтому состояние может обновиться не сразу после вызова setState(). Это означает, что вам не следует полагаться на текущее состояние при вызове setState(), поскольку вы не можете быть уверены в том, каким оно является. Решением названной проблемы является передача в setState() функции с предыдущим состоянием в качестве аргумента. Это позволяет избежать проблем, связанных с получением пользователем старого состояния из-за асинхронной природы setState().

Допустим, начальным значением count является 0. Несмотря на три последовательных вызова операции по увеличению значения, count равняется 1:

// предположим, что this.state.count === 0

this.setState({ count: this.state.count + 1 })

this.setState({ count: this.state.count + 1 })

this.setState({ count: this.state.count + 1 })

// this.state.count === 1, а не 3

Если мы передадим функцию в setState() значение count увеличится корректно:

this.setState((prevState, props) => ({

count: prevState.count + props.increment

}))

// this.state.count === 3 как ожидается

(или)

Почему в setState() рекомендуется использовать функцию, а не объект?

React может объединять несколько вызовов setState() в один для повышения производительности. Поскольку this.props и this.state обновляются асинхронно, вам не следует полагаться на их значения для вычисления следующего состояния.

Приведенный ниже пример счетчика работает неправильно:

// Неправильно

this.setState({

counter: this.state.counter + this.props.increment,

})

Рекомендуемый подход заключается в вызове setState() с функцией в качестве аргумента. Эта функция принимает предыдущее состояние в качестве первого параметра и обновленные пропы в качестве второго параметра:

// Correct

this.setState((prevState, props) => ({

counter: prevState.counter + props.increment

}))

1. Что такое строгий режим в React?

React.StrictMode - это полезный компонент, обеспечивающий индикацию потенциальных проблем в приложении. Как и <Fragment>, <StrictMode> не приводит к рендеренгу лишних DOM-элементов. Он активирует дополнительные проверки и предупреждения для потомков. Эти проверки применяются только в режиме для разработки.

import React from 'react'

function ExampleApplication() {

return (

<div>

<Header />

<React.StrictMode>

<div>

<ComponentOne />

<ComponentTwo />

</div>

</React.StrictMode>

<Footer />

</div>

)

}

В приведенном примере строгий режим включен только для компонентов ComponentOne и ComponentTwo.

1. Что такое React-примеси (Mixins)?

Mixins - это способ обеспечения совместного использования функциональности совершенно разными компонентами. Примеси не следует использовать, их можно заменить компонентами высшего порядка или декораторами.

Одним из самых распространенных случаев использования примесей является PureRenderMixin. Вы можете использовать его в некоторых компонентах для предотвращения повторного рендеринга, когда пропы и состояние поверхностно равны предыдущим:

const PureRenderMixin = require('react-addons-pure-render-mixin')

const Button = React.createClass({

mixins: [PureRenderMixin],

// ...

})

В настоящее время примеси признаны устаревшими.

1. Почему isMounted() является антипаттерном? Назовите более подходящее решение

Основная цель использования isMounted() заключается в предотвращении вызова setState() после размонтирования компонента, поскольку это приводит к выводу в консоль предупреждения.

if (this.isMounted()) {

this.setState({...})

}

Проверка isMounted() перед вызовом setState() отключает предупреждения, но противоречит цели этих предупреждений. Использование isMounted() - это дурно пахнущий код, поскольку единственная причина его использования заключается в предположении, что вам потребуется ссылка на компонент после его размонтирования.

Оптимальным решением является определение места, где setState() может быть вызван после размонтирования компонента, и его удаление. Такие ситуации обычно возникают в коллбеках, когда компонент ожидает получения некоторых данных и размонтируется до их получения. В идеале, все коллбеки должны отключаться в componentWillUnmount() перед размонтированием.

1. Какие события указателя (Pointer Events) поддерживаются в React?

Pointer Events предоставляют унифицированный способ обработки всех событий ввода. Раньше у нас была мышь и соответствующие обработчики, сегодня у нас имеется множество различных устройств, в состав которых мышь не входит, например, телефоны с событиями касаний экрана пальцем или стилусом. Необходимо помнить, что эти события работают только в браузерах, поддерживающих Pointer Events.

Следующие типы событий доступны в React DOM:

* 1. onPointerDown
  2. onPointerMove
  3. onPointerUp
  4. onPointerCancel
  5. onGotPointerCapture
  6. onLostPointerCapture
  7. onPointerEnter
  8. onPointerLeave
  9. onPointerOver
  10. onPointerOut

1. Почему название компонента должно начинаться с большой буквы?

Если вы рендерите компонент с помощью JSX, название этого компонента должно начинаться с большой буквы, в противном случае, React выбросит исключение, связанное с неопознанным тегом. Принятое соглашение гласит, что с маленькой буквы могут начинаться только HTML и SVG-теги.

class SomeComponent extends Component {

// ...

}

Вы можете определить классовый компонент, название которого начинается с маленькой буквы, но при импорте название должно начинаться с большой буквы. Здесь название, начинающееся с маленькой буквы, не вызовет проблем:

class myComponent extends Component {

render() {

return <div />

}

}

export default myComponent

При импорте указанного компонента в другой файл, его название должно начинаться с большой буквы:

import MyComponent from './MyComponent'

Существуют ли исключения из этого правила?

Названия компонентов должны начинаться с большой буквы, за одним исключением. Имена тегов, начинающиеся с маленькой буквы, с точкой (аксессоры) являются валидными названиями компонентов.

Например, следующий тег будет скомпилирован в валидный компонент:

```javascript

render(){

return (

<obj.component /> // `React.createElement(obj.component)`

)

}

```

1. Поддерживаются ли пользовательские DOM-атрибуты в React 16?

Да. В прошлом React игнорировал неизвестные атрибуты. Если вы использовали атрибут, который React не мог распознать, он просто его пропускал:

Например, элемент с таким атрибутом:

<div mycustomattribute={'something'} />

в React 15 превращался в пустой div:

<div />

В React 16 любые неизвестные атрибуты встраиваются в DOM:

<div mycustomattribute='something' />

Это полезно с точки зрения использования специфичных для браузера атрибутов, тестирования новых DOM API и интеграции с другими библиотеками.

1. В чем разница между constructor() и getInitialState()?

При использовании ES6-классов состояние следует инициализировать в конструкторе, а при использовании React.createClass() - в методе getInitialState().

ES6-классы:

class MyComponent extends React.Component {

constructor(props) {

super(props)

this.state = { /\* начальное состояние \*/ }

}

}

React.createClass():

const MyComponent = React.createClass({

getInitialState() {

return { /\* начальное состояние \*/ }

}

})

Обратите внимание: React.createClass() признан устаревшим и удален в React 16. Для создания компонентов с состоянием используйте классы или функции с хуками.

1. Можно ли принудительно обновить компонент без вызова setState()?

По умолчанию компонент перерисовывается при изменении его состояния или пропов. Если метод render() зависит от других данных, вы можете сообщить React о том, что компонент нуждается в повторном рендеринге, вызвав forceUpdate().

component.forceUpdate(callback)

Использовать forceUpdate() не рекомендуется.

1. В чем разница между super() и super(props) в классовых компонентах React?

Если вы хотите получить доступ к this.props в constructor(), тогда пропы должны быть переданы в метод super().

super(props):

class MyComponent extends React.Component {

constructor(props) {

super(props)

console.log(this.props) // { name: 'John', ... }

}

}

super():

class MyComponent extends React.Component {

constructor(props) {

super()

console.log(this.props) // undefined

}

}

За пределами constructor(), this.props имеет одинаковое значение.

1. Как реализовать цикл внутри JSX?

Для этого можно использовать Array.prototype.map() и стрелочную функцию.

Например, в следующем примере массив объектов items преобразуется в массив компонентов:

<tbody>

{items.map(item => <SomeComponent key={item.id} name={item.name} />)}

</tbody>

Однако, с помощью for реализовать цикл не получится:

<tbody>

for (let i = 0; i < items.length; i++) {

<SomeComponent key={items[i].id} name={items[i].name} />

}

</tbody>

Это объясняется тем, что JSX-теги транспилируются в вызовы функций, а мы не можем использовать операторы внутри выражений. Ситуация может измениться благодаря выражению do, которое находится на 1 стадии рассмотрения.

1. Как получить доступ к пропам в закавыченных значениях атрибутов?

React (или JSX) не поддерживает интерполяцию переменных в значениях атрибутов. Следующий код не будет работать:

<img className='image' src='images/{this.props.image}' />

Однако, вы можете поместить любое JavaScript-выражение в фигурные скобки как входящее значение атрибута. Следующий код работает, как ожидается:

<img className='image' src={'images/' + this.props.image} />

Также можно использовать шаблонные литералы (template literals):

<img className='image' src={`images/${this.props.image}`} />

1. Что такое массив React PropType с формой (shape)?

Если вы хотите передать массив объектов в компонент с определенной формой, тогда используйте React.PropTypes.shape() в качестве аргумента React.PropTypes.arrayOf():

ReactComponent.propTypes = {

arrayWithShape: React.PropTypes.arrayOf(React.PropTypes.shape({

color: React.PropTypes.string.isRequired,

fontSize: React.PropTypes.number.isRequired

})).isRequired

}

1. Как реализовать условное применение классовых атрибутов?

Вам не следует использовать фигурные скобки внутри кавычек, поскольку в этом случае они будут оцениваться как строка:

<div className="btn-panel {this.props.visible ? 'show' : 'hidden'}">

Вместо этого, фигурные скобки следует поместить снаружи (не забудьте добавить пробел между названиями классов):

<div className={'btn-panel ' + (this.props.visible ? 'show' : 'hidden')}>

Либо используйте шаблонные строки:

<div className={`btn-panel ${this.props.visible ? 'show' : 'hidden'}`}>

1. В чем разница между React и ReactDOM?

Пакет react содержит React.createElement(), React.Component, React.Children и другие вспомогательные функции, связанные с элементами и компонентами. Вы можете думать о них как об изоморфных или универсальных помощниках в создании компонентов. Пакет react-dom содержит ReactDOM.render(), а в react-dom/server у нас имеется рендеринг на стороне сервера, обеспечиваемый такими методами как ReactDOMServer.renderToString() и ReactDOMServer.renderToStaticMarkup().

1. Почему ReactDOM отделен от React?

Команда React проделала большую работу по извлечению всех "фич", связанных с DOM, в отдельную библиотеку под названием ReactDOM. Впервые библиотеки были разделены в React 0.14. Учитывая другие библиотеки, такие как react-native, react-art, react-canvas и react-three, становится очевидным, что React не должен быть тесно связан с браузером или DOM.

Для обеспечения рендеринга в разных средах, команда React разделила основной пакет React на две части: react и react-dom. Это позволяет создавать компоненты, которые могут использоваться как в веб, так и в мобильной версиях приложения.

1. Как использовать элемент label в React?

Если вы попытаетесь отрендерить элемент label, привязав его к инпуту с помощью стандартного атрибута for, то в готовой разметке этот атрибут будет пропущен, а в консоль будет выведено предупреждение:

<label for='user'>Пользователь</label>

<input type='text' id='user' />

Поскольку for в JavaScript является ключевым словом, вместо него следует использовать htmlFor:

<label htmlFor='user'>Пользователь</label>

<input type='text' id='user' />

1. Как совместно использовать несколько встроенных объектов со стилями?

Для этого можно использовать spread-оператор в обычном React:

<button style={{...styles.panel.button, ...styles.panel.submitButton}}>Отправить</button>

В React Native можно использовать синтаксис массивов:

<button style={[styles.panel.button, styles.panel.submitButton]}>Отправить</button>

1. Как повторно отрендерить слой представления при изменении размеров браузера?

Для этого можно прослушивать событие resize в методе componentDidMount() и обновлять направления (width и height). Не забудьте удалить обработчик в методе componentWillUnmount().

class WindowDimensions extends React.Component {

constructor(props){

super(props);

this.updateDimensions = this.updateDimensions.bind(this);

}

componentWillMount() {

this.updateDimensions()

}

componentDidMount() {

window.addEventListener('resize', this.updateDimensions)

}

componentWillUnmount() {

window.removeEventListener('resize', this.updateDimensions)

}

updateDimensions() {

this.setState({width: window.innerWidth, height: window.innerHeight})

}

render() {

return <span>{this.state.width} x {this.state.height}</span>

}

}

Вот аналогичный функциональный компонент:

function WindowDimensions() {

const [state, setState] = useState({

width: window.innerWidth,

height: window.innerHeight

})

function updateDimensions() {

setState({

width: window.innerWidth,

height: window.innerHeight

})

}

useEffect(() => {

window.addEventListener('resize', updateDimensions)

return () => {

window.removeEventListener('resize', updateDimensions)

}

}, [])

return <span>{state.width} x {state.height}</span>

}

1. В чем разница между методами setState() и replaceState()?

При использовании setState() текущее и предыдущее состояния объединяются. replaceState() удаляет текущее состояние и заменяет его переданным. Обычно, setState() используется до тех пор, пока вам действительно не понадобится удалить все предыдущие ключи по какой-то причине. Вы также можете установить состояние в значение false/null в setState() вместо использования replaceState().

1. Как следить за изменением состояния компонента?

При обновлении состояния вызывается метод жизненного цикла componentDidUpdate(). Вы можете сравнить передаваемое состояние и пропы с предыдущими для выявления изменений.

componentDidUpdate(object prevProps, object prevState)

Обратите внимание: в предыдущих версиях React для обновления состояния можно было использовать componentWillUpdate(object nextProps, object nextState). Впоследствии, данный метод был признан устаревшим.

1. Как рекомендуется удалять элемент из массива в состоянии React?

Оптимальный подход заключается в использовании метода Array.prototype.filter().

Создадим метод removeItem() для обновления состояния:

removeItem(index) {

this.setState({

data: this.state.data.filter((item, i) => i !== index)

})

}

1. Возможно ли использовать React без рендеринга HTML?

В новых версиях (>=16.2) это вполне возможно. Ниже представлено несколько вариантов:

render() {

return false

}

render() {

return null

}

render() {

return []

}

render() {

return <React.Fragment></React.Fragment>

}

render() {

return <></>

}

С undefined это не работает.

1. Как красиво отобразить JSON с помощью React?

Мы можем использовать тег pre для сохранения форматирования, произведенного JSON.stringify():

const data = { name: 'Иван', age: 42 }

class User extends React.Component {

render() {

return (

<pre>

{JSON.stringify(data, null, 2)}

</pre>

)

}

}

React.render(<User />, document.getElementById('container'))

1. Почему в React нельзя обновлять пропы?

Философия React гласит, что пропы должны быть иммутабельными (неизменяемыми или неизменными) и однонаправленными (передаваемыми в одном направлении, сверху вниз). Это означает, что родительский компонент может передавать пропы дочерним, а последние не могут их модифицировать.

1. Как установить фокус на инпут при загрузке страницы?

Это можно сделать, создав ссылку (ref) на элемент input, и использовав метод componentDidMount():

class App extends React.Component{

componentDidMount() {

this.nameInput.focus()

}

render() {

return (

<div>

<input

defaultValue='Не в фокусе'

/>

<input

ref={(input) => this.nameInput = input}

defaultValue='В фокусе'

/>

</div>

)

}

}

ReactDOM.render(<App />, document.getElementById('app'))

1. Какие существуют способы обновления объекта состояния?
   1. Вызов setState() с объектом, объединяемым с состоянием:
      * Использование Object.assign() для создания копии объекта:

const user = Object.assign({}, this.state.user, { age: 42 })

this.setState({ user })

* + - Использование spread-оператора:

const user = { ...this.state.user, age: 42 }

this.setState({ user })

* 1. Вызов setState() с функцией:

this.setState(prevState => ({

user: {

...prevState.user,

age: 42

}

}))

* 1. Вызов сеттера, возвращенного useState():

const [user, setUser] = useState({name: 'Иван'})

setUser({...user, age: 42})

1. Как получить версию React при запуске приложения в браузере?

Для этого можно использовать React.version:

const REACT\_VERSION = React.version

ReactDOM.render(

<div>{`Версия React: ${REACT\_VERSION}`}</div>,

document.getElementById('app')

)

1. Какие существуют подходы к добавлению полифилов в create-react-app?
   1. Импорт из core-js вручную:

Создаем файл, например, polyfills.js и импортируем его в корневой файл index.js. Выполняем npm install core-js или yarn add core-js и импортируем необходимый функционал:

import 'core-js/fn/array/find'

import 'core-js/fn/array/includes'

import 'core-js/fn/number/is-nan'

* 1. Использование специального сервиса:

Извлекаем из polyfill.io CDN определенные, специфичные для браузера полифилы посредством добавления такой строки в index.html:

<script src='https://cdn.polyfill.io/v2/polyfill.min.js?features=default,Array.prototype.includes'></script>

В примере мы дополнительно запрашиваем Array.prototype.includes(), поскольку он не включен в стандартный набор.

1. Как использовать https вместо http в create-react-app?

Для этого нужно использовать опцию HTTPS=true. Вы можете отредактировать раздел со скриптами в package.json следующим образом:

"scripts": {

"start": "set HTTPS=true && react-scripts start"

}

Или просто выполнить команду set HTTPS=true && npm start.

1. Как избежать использования относительных путей при импорте в create-react-app?

Создайте файл .env в корневой директории проекта и добавьте в него адрес импорта:

NODE\_PATH=src/app

Затем перезагрузите сервер для разработки. После этого у вас появится возможность импортировать все, что находится в src/app, без использования относительных путей.

Вы также можете создать в корневой директории проекта файл jsconfig.json следующего содержания:

{

"compilerOptions": {

"baseUrl": "src"

},

"include": [

"src"

]

}

В последнем случае импорт файлов будет начинаться с src:

// вместо '../app', например

import Component from 'app/Component.js'

1. Как добавить Google Analytics в React Router?

Добавьте обработчик к объекту history для записи каждого отображения страницы:

history.listen(function (location) {

window.ga('set', 'page', location.pathname + location.search)

window.ga('send', 'pageview', location.pathname + location.search)

})

1. Как обновлять состояние компонента каждую секунду?

Для этого нужно использовать setInterval(), изменяющий состояние. Не забудьте отключить таймер при размонтировании компонента во избежание ошибок и утечек памяти:

componentDidMount() {

this.interval = setInterval(() => this.setState({ time: Date.now() }), 1000)

}

componentWillUnmount() {

clearInterval(this.interval)

}

Аналогичный функциональный компонент:

```javascript

const [time, setTime] = useState(Date.now)

useEffect(() => {

const timerId = setInterval(() => setTime(Date.now), 1000)

return () => {

clearInterval(timerId)

}

}, [time])

```

1. Как применить вендорные префиксы во встроенных стилях в React?

React не применяет вендорные префиксы автоматически. Вам необходимо добавлять их вручную:

<div style={{

transform: 'rotate(90deg)',

WebkitTransform: 'rotate(90deg)', // обратите внимание на большую букву "W"

msTransform: 'rotate(90deg)' // 'ms' - это единственный вендорный префикс в нижнем регистре

}} />

Такие библиотеки, как styled-components добавляют префиксы автоматически.

1. Как экспортировать/импортировать компоненты с помощью React и ES6?

Для экспорта компонентов рекомендуется использовать экспорт по умолчанию:

import React from 'react'

import User from 'user'

export default class MyProfile extends React.Component {

render(){

return (

<User type="customer">

//...

</User>

)

}

}

Вспомогательные элементы могут экспортироваться по названию.

Обратите внимание: при агрегации экспортируемых компонентов в файле index.js, удобнее использовать именованный экспорт, в противном случае, придется делать так:

export { default as Component } from './Component'

1. Почему конструктор компонента вызывается только один раз?

Алгоритм согласования без дополнительной информации предполагает, что если определенный компонент находится на том же месте при последующем рендеринге, то это тот же самый компонент, что и раньше, поэтому используется предыдущий экземпляр вместо создания нового.

1. Как в React определяются константы?

Для определения константы можно использовать синтаксис статических полей класса:

class MyComponent extends React.Component {

static DEFAULT\_PAGINATION = 10

}

Данное предложение находится на 3 стадии рассмотрения.

1. Как программно вызвать возникновение события клика в React?

Вы можете использовать проп ref через коллбек в качестве ссылки на HTMLInputElement, сохранить эту ссылку в свойстве класса и использовать ее для вызова события в обработчике с помощью метода click().

Это делается в два этапа:

* 1. Создаем ссылку в методе render():

<input ref={input => this.inputElement = input} />

* 1. Вызываем событие клика в обработчике:

this.inputElement.click()

Тоже самое можно реализовать с помощью хука useRef().

1. Можно ли использовать async/await в обычном React?

Новые версии React поддерживают синтаксис async/await из коробки. Раньше для этого надо было использовать Babel и плагин [transform-async-to-generator](https://babeljs.io/docs/en/babel-plugin-transform-async-to-generator).

1. Назовите общую структуру директорий в React-проекте

Существует два варианта структурирования файлов проекта на React:

* 1. Группировка по "фичам" или маршрутам:

common/

├─ Avatar.js

├─ Avatar.css

├─ APIUtils.js

└─ APIUtils.test.js

feed/

├─ index.js

├─ Feed.js

├─ Feed.css

├─ FeedStory.js

├─ FeedStory.test.js

└─ FeedAPI.js

profile/

├─ index.js

├─ Profile.js

├─ ProfileHeader.js

├─ ProfileHeader.css

└─ ProfileAPI.js

* 1. Группировка по типам файлов:

api/

├─ APIUtils.js

├─ APIUtils.test.js

├─ ProfileAPI.js

└─ UserAPI.js

components/

├─ Avatar.js

├─ Avatar.css

├─ Feed.js

├─ Feed.css

├─ FeedStory.js

├─ FeedStory.test.js

├─ Profile.js

├─ ProfileHeader.js

└─ ProfileHeader.css

1. Назовите популярные библиотеки для работы с анимацией в React

React Transition Group и React Motion являются самыми популярными библиотеками для работы с анимацией в экосистеме React.

1. В чем заключаются преимущества использования модулей со стилями?

Рекомендуется избегать жесткого кодирования стилей в компонентах. Любые значения, которые предполагается использовать в нескольких компонентах, должны быть извлечены в собственные модули.

Например, эти стили могут быть извлечены в разные компоненты:

export const colors = {

white,

black,

blue

}

export const space = [

0,

8,

16,

32,

64

]

И затем импортироваться в компоненты по необходимости:

import { space, colors } from './styles'

1. Назовите популярные линтеры для React

Самым популярным линтером для JavaScript является ESLint. Для анализа специфического кода доступны специальные плагины. Одним из наиболее распространенных в случае с React является пакет eslint-plugin-react. По умолчанию он следует набору лучших практик, включая правила определения наличия ключей в циклах и полный набор типов пропов.

Другим популярным инструментом является eslint-plugin-jsx-a11y, который помогает исправлять распространенные проблемы с доступностью. Поскольку JSX отличается от обычного HTML, проблемы с текстом alt или tabindex, например, не будут обнаруживаться обычными плагинами.

1. Как выполнить AJAX-запрос и в каком методе жизненного цикла это следует делать?

Для этого можно использовать такие библиотеки, как jQuery AJAX, Axios или встроенный fetch. Вы должны запрашивать данные в методе componentDidMount(). Далее для обновления компонента (при получении данных) используется setState().

В следующем примере список сотрудников запрашивается из API и записывается в локальное состояние:

class MyComponent extends React.Component {

constructor(props) {

super(props)

this.state = {

employees: [],

error: null

}

}

componentDidMount() {

fetch('https://api.example.com/items')

.then(res => res.json())

.then(

(result) => {

this.setState({

employees: result.employees

})

},

(error) => {

this.setState({ error })

}

)

}

render() {

const { error, employees } = this.state

if (error) {

return <div>Error: {error.message}</div>;

} else {

return (

<ul>

{employees.map(employee => (

<li key={employee.name}>

{employee.name}-{employee.experience}

</li>

))}

</ul>

)

}

}

}

Тоже самое можно реализовать с помощью хука `useEffect()` и сеттера, возвращённого хуком `useState()`.

1. Что такое рендер-пропы (Render Props)?

Render Props - это техника распределения кода между компонентами с помощью пропа, чьим значением является функция. Следующий компонент использует рендер-проп, возвращающий React-элемент:

<DataProvider render={data => (

<h1>{`Привет, ${data.target}`}</h1>

)}/>

Этот паттерн широко используется такими библиотеками, как React Router и DownShift.

## React Router

### Что такое React Router?

React Router - это мощная библиотека маршрутизации, построенная на основе React, которая помогает добавлять новые страницы и перемещаться между ними очень быстро, при сохранении синхронизации между адресом страницы и ее представлением.

### Чем React Router отличается от библиотеки history?

React Router - это обертка библиотеки history, которая обеспечивает взаимодействие с браузерным window.history и хэш-историей. Она также позволяет сохранять историю в памяти при отсутствии в окружении глобальной истории, например, при разработке мобильных приложений (React Native) и юнит-тестировании с помощью Node.

### Что такое Router в React Router 4?

React Router 4 предоставляет 3 компонента Router:

* 1. BrowserRouter
  2. HashRouter
  3. MemoryRouter

Указанные компоненты создают экземпляры browser, hash и memory истории. React Router 4 делает свойства и методы экземпляра history связанными с вашим мартшрутизатором через контекст в объекте router.

### Для чего предназначены методы push() и replace() объекта History?

Экземпляр истории имеет два метода для навигации:

* 1. push()
  2. replace()

Если думать об истории как о массиве посещенных локаций, то метод push() добавляет в массив новую локацию, а replace() заменяет текущую локацию новой.

### Как реализовать программную навигацию с помощью React Router 4?

Существует, как минимум, три способа реализовать программный роутинг/навигацию в компоненте:

* 1. **Использование функции высшего порядка**withRouter()**:**

withRouter() внедряет объект истории в качестве пропа в компонент. Этот объект предоставляет методы push() и replace(), позволяющие избежать использования контекста:

import { withRouter } from 'react-router-dom' // это также работает с 'react-router-native'

const Button = withRouter(({ history }) => (

<button

type='button'

onClick={() => { history.push('/new-location') }}

>

Нажми на меня!

</button>

))

* 1. **Использование компонента**Route**и рендер-пропов:**

Компонент Route предоставляет такой же проп, что и withRouter(), так что у вас будет возможность получить доступ к методам истории через проп history.

import { Route } from 'react-router-dom'

const Button = () => (

<Route render={({ history }) => (

<button

type='button'

onClick={() => { history.push('/new-location') }}

>

Нажми на меня!

</button>

)} />

)

* 1. **Использование контекста:**

Данный подход использовать не рекомендуется, он считается нестабильным:

const Button = (props, context) => (

<button

type='button'

onClick={() => {

context.history.push('/new-location')

}}

>

Нажми на меня!

</button>

)

Button.contextTypes = {

history: React.PropTypes.shape({

push: React.PropTypes.func.isRequired

})

}

### Как получить параметры строки запроса в React Router 4?

Возможность разбирать строку запроса была удалена из React Router 4, поскольку разработчики на протяжении нескольких лет просили добавить другую реализацию. Поэтому разработчикам была предоставлена возможность самостоятельного выбора реализации. Рекомендуемым подходом является использование библиотеки query-srting:

const queryString = require('query-string');

const parsed = queryString.parse(props.location.search);

Если вы предпочитаете нативные решения, то можете использовать URLSearchParams():

const params = new URLSearchParams(props.location.search)

const foo = params.get('name')

### Когда можно получить предупреждение "Router может содержать только один дочерний элемент"?

Маршруты (routes) должны быть обернуты в блок Switch (переключатель), поскольку этот блок является уникальным и рендерит только компонент по определенному маршруту.

Прежде всего, необходимо импортировать Switch:

import { Switch, Router, Route } from 'react-router'

Затем определить в нем маршруты:

<Router>

<Switch>

<Route {/\* ... \*/} />

<Route {/\* ... \*/} />

</Switch>

</Router>

### Как передать параметры методу history.push() в React Router 4?

В процессе навигации вы можете передавать объекту history некоторые пропы:

this.props.history.push({

pathname: '/template',

search: '?name=sudheer',

state: { detail: response.data }

})

Свойство search, например, используется для передачи параметров строки запроса.

### Как реализовать стандартную или NotFound страницу?

Switch рендерит компонент по первому совпавшему маршруту. Route без пути всегда будет совпадать. Поэтому можно просто опустить атрибут path или же указать /\* в качестве его значения:

<Switch>

<Route exact path="/" component={Home}/>

<Route path="/user" component={User}/>

<Route component={NotFound} />

</Switch>

### Как получить объект истории в React Router 4?

Для получения объекта истории необходимо выполнить следующие шаги:

* 1. Создать модуль, экспортирующий объект history, и импортировать этот модуль в проект.

Создаем файл history.js:

import { createBrowserHistory } from 'history'

export default createBrowserHistory({

/\* здесь может быть объект с настройками \*/

})

* 1. Вместо встроенных роутеров следует использовать компонент Router. Импортируем history в файл index.js:

import { Router } from 'react-router-dom'

import history from './history'

import App from './App'

ReactDOM.render((

<Router history={history}>

<App />

</Router>

), holder)

* 1. Вы также можете использовать метод push() объекта history по аналогии со встроенным объектом истории:

// some-other-file.js

import history from './history'

history.push('/go-here')

### Как реализовать автоматическое перенаправление после выполнения входа в систему?

Пакет react-router предоставляет компонент Redirect. Рендеринг данного компонента приводит к перемещению в новую локацию. Как и в случае с серверными перенаправлениями, новая локация перезапишет текущую в стеке истории:

import React, { Component } from 'react'

import { Redirect } from 'react-router'

export default class LoginComponent extends Component {

render() {

if (this.state.isLoggedIn === true) {

return <Redirect to="/your/redirect/page" />

} else {

return <div>Войдите, пожалуйста</div>

}

}

}

## Интернационализация в React

### Что такое React Intl?

React Intl - библиотека, облегчающая работу с интернационализацией в React, включая готовые компоненты и API для обработки строк, чисел, дат и т.д. React Intl является адаптированной версией FormatJS.

### Назовите основные возможности React Intl

React Intl предоставляет следующие возможности:

* 1. Отображение чисел с разделителями
  2. Правильное отображение даты и времени
  3. Отображение разницы между датой и "сейчас"
  4. Плюрализация подписей в строках
  5. Поддержка 150+ языков
  6. Возможность запуска как в браузере, так и в Node
  7. Следование стандартам

### Назовите основные способы форматирования в React Intl

Библиотека React Intl предоставляет два способа форматирования строк, чисел и дат:

* 1. **С помощью компонентов React:**

<FormattedMessage

id='account'

defaultMessage='На балансе недостаточно средств.'

/>

2. \*\*С помощью API:\*\*

```javascript

const messages = defineMessages({

accountMessage: {

id: 'account',

defaultMessage: 'На балансе недостаточно средств.',

}

})

formatMessage(messages.accountMessage)

```

### Как использовать FormattedMessage в качестве заместителя в React Intl?

Компонент Formatted из react-intl возвращает элементы, а не обычный текст, которые не могут использоваться в качестве заместителей (placeholders), альтернативного текста и т.д. Для этого необходимо использовать низкоуровневое API formatMessage(). Объект intl внедряется в компонент с помощью компонента высшего порядка injectIntl(), затем сообщение форматируется с помощью formatMessage(), доступного в этом объекте:

import React from 'react'

import { injectIntl, intlShape } from 'react-intl'

const MyComponent = ({ intl }) => {

const placeholder = intl.formatMessage({id: 'messageId'})

return <input placeholder={placeholder} />

}

MyComponent.propTypes = {

intl: intlShape.isRequired

}

export default injectIntl(MyComponent)

### Как получить текущую локализацию с помощью React Intl?

Текущую локализацию в любом компоненте приложения можно получить с помощью injectIntl():

import { injectIntl, intlShape } from 'react-intl'

const MyComponent = ({ intl }) => (

<div>{`Текущая локализация: ${intl.locale}`}</div>

)

MyComponent.propTypes = {

intl: intlShape.isRequired

}

export default injectIntl(MyComponent)

### Как отформатировать дату с помощью React Intl?

Компонент высшего порядка injectIntl() предоставляет доступ к методу formatDate() через пропы компонента. Этот метод используется внутренним механизмом экземпляров FormattedDate и возвращает строковое представление отформатированной даты:

import { injectIntl, intlShape } from 'react-intl'

const stringDate = this.props.intl.formatDate(date, {

year: 'numeric',

month: 'numeric',

day: 'numeric'

})

const MyComponent = ({intl}) => (

<div>{`Отформатированная дата: ${stringDate}`}</div>

)

MyComponent.propTypes = {

intl: intlShape.isRequired

}

export default injectIntl(MyComponent)

## Тестирование в React

### Что такое поверхностный рендеринг (Shallow Rendering) в терминах тестирования?

Shallow rendering используется для юнит-тестирования в React. Он позволяет рендерить плоский компонент и оценивать результаты, возвращаемые его методом render(), не заботясь о поведении дочерних компонентов, которые в момент тестирования еще не инстанцированы или не отрисованы.

Например, если у нас имеется такой компонент:

function MyComponent() {

return (

<div>

<span className={'heading'}>{'Заголовок'}</span>

<span className={'description'}>{'Описание'}</span>

</div>

)

}

Мы можем протестировать его следующим образом

import ShallowRenderer from 'react-test-renderer/shallow'

// код теста

const renderer = new ShallowRenderer()

renderer.render(<MyComponent />)

const result = renderer.getRenderOutput()

expect(result.type).toBe('div')

expect(result.props.children).toEqual([

<span className='heading'>Заголовок</span>,

<span className='description'>Описание</span>

])

### Что такое TestRenderer в React?

Данный пакет предоставляет движок, который используется для рендеринга компонентов в виде "чистых" объектов JavaScript, не связанных с DOM или мобильным окружением. Он позволяет получить снимок иерархии компонентов (похожий на DOM-дерево), генерируемых ReactDOM или React Native без помощи браузера или jsdom:

import TestRenderer from 'react-test-renderer'

const Link = ({page, children}) => <a href={page}>{children}</a>

const testRenderer = TestRenderer.create(

<Link page={'https://www.facebook.com/'}>Facebook</Link>

)

console.log(testRenderer.toJSON())

// {

// type: 'a',

// props: { href: 'https://www.facebook.com/' },

// children: [ 'Facebook' ]

// }

### Для чего предназначен ReactTestUtils?

ReactTestUtils, входящий в состав пакета with-addons, позволяет имитировать поведение DOM для целей юнит-тестирования.

### Что такое Jest?

Jest - это JavaScript-фреймворк для юнит-тестирования, созданный Facebook на основе Jasmine, предоставляющий возможность автоматического создания "моков" (mocks) и окружение jsdom. Он часто используется для тестирования компонентов.

### В чем заключаются преимущества Jest перед Jasmine?

Таких преимуществ несколько:

* + Автоматическое выполнение тестов из исходного кода
  + Автоматическое внедрение зависимостей при выполнении тестов
  + Возможность синхронного тестирования асинхронного кода
  + Тесты запускаются в "фейковом" DOM (с помощью jsdom), что позволяет запускать тесты из командной строки
  + Одновременный запуск нескольких тестов, что позволяет им выполняться быстрее

### Приведите простой пример использования Jest

Создадим тест для функции, возвращающей сумму двух чисел, находящейся в файле sum.js:

const sum = (a, b) => a + b

export default sum

Создаем файл sum.test.js, содержащий код теста:

import sum from './sum'

test('сумма 1 и 2 равняется 3', () => {

expect(sum(1, 2)).toBe(3)

})

Затем добавляем в package.json следующую строку:

{

"scripts": {

"test": "jest"

}

}

Наконец, выполняем yarn test или npm test, и Jest покажет результат теста:

$ yarn test

PASS ./sum.test.js

✓ сумма 1 и 2 равняется 3 (2ms)

## React Redux

### Что такое Flux?

Flux - это парадигма проектирования приложений, являющаяся альтернативой более традиционному паттерну MVC. Это не фреймворк или библиотека, а новый вид архитектуры, разработанный специально для React с учетом концепции однонаправленного потока данных. Facebook использует данный паттерн в своих внутренних проектах.

Рабочий процесс или взаимодействие компонентов диспетчера (dispatcher), хранилища (store) и представления (view) с учетом каналов ввода-вывода выглядит так

### Что такое Redux?

Redux - это стабильный (предсказуемый) контейнер для хранения состояния JavaScript-приложений, основанный на паттерне проектирования Flux. Redux может использоваться с React и любой другой библиотекой. Он легкий (около 2 Кб) и не имеет зависимостей.

### Назовите ключевые принципы Redux

Redux следует трем фундаментальным принципам:

* 1. **Единственный источник истины:** состояние всего приложения хранится в древовидном объекта - в одном хранилище. Единственное состояние-дерево облегчает наблюдение за изменениями и отладку или инспектирование приложения
  2. **Состояние доступно только для чтения:** единственный способ изменить состояние заключается в запуске операции - объекте, описывающем произошедшее. Это позволяет гарантировать, что ни представления, ни сетевые коллбеки не буду иметь возможности изменять состояние напрямую
  3. **Изменения производятся с помощью "чистых" функций:** для определения того, как изменяется состояние в зависимости от операции, создаются редукторы (reducers). Редукторы - это "чистые" функции, принимающие предыдущее состояние в качестве аргумента и возвращающие новое

### Проведите сравнение Redux и Flux

Отличия между Redux и Flux можно свести к следующему:

* 1. **Недопустимость мутаций:** во Flux состояние может быть изменяемым, а Redux требует, чтобы состояние было иммутабельным, и многие библиотеки для Redux исходят из предположения, что вы никогда не будете менять состояние напрямую. Вы можете обеспечить иммутабельность состояния с помощью таких пакетов, как redux-immutable-state-invariant, Immutable.js или условившись с другими членами команды о написании иммутабельного кода
  2. **Осторожность в выборе библиотек:** Flux не пытается решать такие проблемы, как повторное выполнение/отмена выполнения, стабильность (постоянство) кода или проблемы, связанные с обработкой форм, явно, а Redux имеет возможность к расширению с помощью промежуточного программного обеспечения (middleware) и предохранителей хранилища, что породило богатую экосистему
  3. **Отсутствие интеграции с Flow:** Flux позволяет осуществлять очень выразительную статическую проверку типов, а Redux пока не поддерживает такой возможности

### В чем разница между mapStateToProps() и mapDispatchToProps()?

mapStateToProps() - это утилита, помогающая компонентам получать обновленное состояние (которое было обновлено другим компонентом):

const mapStateToProps = (state) => {

return {

todos: getVisibleTodos(state.todos, state.visibilityFilter)

}

}

mapDispatchToProps() - утилита, помогающая компонентам вызывать операции (которые могут привести к обновлению состояния приложения):

const mapDispatchToProps = (dispatch) => {

return {

onTodoClick: (id) => {

dispatch(toggleTodo(id))

}

}

}

Для mapDispatchToProps() рекомендуется всегда использовать короткую форму записи объекта.

Redux оборачивает ее в другую функцию, которая выглядит как (…args) => dispatch(onTodoClick(…args)), и передает эту обертку в качестве пропа в компонент:

const mapDispatchToProps = ({

onTodoClick

})

### Можно ли запускать операцию в редукторе?

Запуск операции в редукторе является антипаттерном. Редуктор не должен вызывать побочных эффектов, он должен принимать объект с названием операции и полезной нагрузкой (payload) и возвращать объект с новым состоянием. Добавление обработчиков и запуск операций в редукторе могут привести к цепной реакции и другим негативным последствиям.

### Как получить доступ к хранилищу Redux за пределами компонента?

Для этого нужно экспортировать хранилище из модуля, в котором оно создано с помощью createStore(). Имейте ввиду, что оно не должно загрязнять глобальное пространство имен:

store = createStore(myReducer)

export default store

### Назовите недостатки паттерна MVW (Model-View-Whatever: MVC, MVP, MVVM и т.д.)

* 1. Манипуляции с DOM являются очень дорогостоящими, что делает приложение медленным и неэффективным
  2. Для обеспечения возможности использования обратных зависимостей была разработана очень сложная модель вокруг моделей и представлений
  3. Происходит изменение большого количества данных в приложениях для совместной работы (таких как Google Docs)
  4. Не существует простого способа отменять действия без добавления большого количества дополнительного кода

### Существует ли что-либо общее между Redux и RxJS?

Названные библиотеки созданы для разных целей. Однако, между ними есть кое-что общее.

Redux - это инструмент для управления состоянием приложения. Как правило, он используется в качестве архитектурного решения для пользовательского интерфейса. Думайте о нем как об альтернативе (наполовину) Angular. RxJS - это библиотека реактивного программирования. Обычно, она используется для выполнения асинхронных задач в JavaScript. Думайте о ней как об альтернативе промисам. Redux использует реактивную парадигму, поскольку хранилище является реактивным. Хранилище наблюдает за операциями с расстояния и изменяет себя. RxJS следует той же парадигме, но вместо того, чтобы выступать в роли архитектуры, он предоставляет основные строительные блоки, наблюдаемые объекты (observables), для реализации указанного паттерна.

### Как запустить операцию при загрузке?

Вы можете запускать операцию в методе componentDidMount() и проверять данные в методе render():

class App extends Component {

componentDidMount() {

this.props.fetchData()

}

render() {

return this.props.isLoaded

? <div>Загружено</div>

: <div>Не загружено</div>

}

}

const mapStateToProps = (state) => ({

isLoaded: state.isLoaded

})

const mapDispatchToProps = { fetchData }

export default connect(mapStateToProps, mapDispatchToProps)(App)

### Как использовать метод connect() в Redux?

Для того, чтобы иметь возможность использовать хранилище в контейнере, необходимо выполнить следующие шаги:

* 1. **Использовать метод**mapStateToProps()**:** он передает переменные состояния из хранилища в определенные вами пропы
  2. **Подключить пропы к контейнеру:** объект, возвращенный mapStateToProps() подключается к контейнеру. Вы можете импортировать connect() из react-redux:

import React from 'react'

import { connect } from 'react-redux'

class App extends React.Component {

render() {

return <div>{this.props.containerData}</div>

}

}

function mapStateToProps(state) {

return { containerData: state.data }

}

export default connect(mapStateToProps)(App)

### Как обнулить состояние в Redux?

Для этого необходимо создать корневой редуктор (root reducer), делегирующий обработку операций редуктору, генерируемому методом combineReducers().

Создадим rootReducer(), возвращающий начальное состояние после операции USER\_LOGOUT. Как мы знаем, редукторы возвращают начальное состояние при вызове с undefined в качестве первого аргумента, независимо от операции:

const appReducer = combineReducers({

/\* редукторы верхнего уровня приложения \*/

})

const rootReducer = (state, action) => {

if (action.type === 'USER\_LOGOUT') {

state = undefined

}

return appReducer(state, action)

}

В случае использования redux-persist, вам, возможно, также потребуется очистить хранилище. redux-persist сохраняет копию состояния в движке хранилища. Поэтому сначала нужно импортировать соответствующий движок, затем разобрать состояние перед установкой его значение в undefined и, наконец, очистить каждый ключ состояния хранилища:

const appReducer = combineReducers({

/\* редукторы верхнего уровня приложения \*/

})

const rootReducer = (state, action) => {

if (action.type === 'USER\_LOGOUT') {

Object.keys(state).forEach(key => {

storage.removeItem(`persist: ${key}`)

})

state = undefined

}

return appReducer(state, action)

}

### Для чего используется символ @ в декораторе connect Redux?

Символ @ указывает на то, что мы имеем дело с декоратором JavaScript. Decorators делают возможным аннотирование и модификацию классов, их полей и методов во время определения класса.

Рассмотрим примеры настройки Redux без и с использованием декоратора:

* + **Без декоратора:**

import React from 'react'

import \* as actionCreators from './actionCreators'

import { bindActionCreators } from 'redux'

import { connect } from 'react-redux'

function mapStateToProps(state) {

return { todos: state.todos }

}

function mapDispatchToProps(dispatch) {

return { actions: bindActionCreators(actionCreators, dispatch) }

}

class MyApp extends React.Component {

// ...

}

export default connect(mapStateToProps, mapDispatchToProps)(MyApp)

* + **C декоратором:**

import React from 'react'

import \* as actionCreators from './actionCreators'

import { bindActionCreators } from 'redux'

import { connect } from 'react-redux'

function mapStateToProps(state) {

return { todos: state.todos }

}

function mapDispatchToProps(dispatch) {

return { actions: bindActionCreators(actionCreators, dispatch) }

}

@connect(mapStateToProps, mapDispatchToProps)

export default class MyApp extends React.Component {

// ...

}

Приведенные примеры почти идентичны, за исключением использования декоратора. Синтаксис декораторов пока не стандартизирован, является экспериментальным и может измениться в будущем (данное предложение находится на 3 стадии рассмотрения). Для поддержки декораторов можно использовать Babel.

### В чем разница между контекстом React и React Redux?

Вы можете использовать Context напрямую, он отлично подходит для передачи данных глубоко вложенным компонентам - в этом состоит его основное назначение.

Redux - гораздо более мощный инструмент, предоставляющий большое количество возможностей, которыми не обладает API контекста. На самом деле, Redux использует контекст в своих внутренних механизмах, но не экстраполирует его в открытый интерфейс.

### Почему функции, изменяющие состояние, в Redux называются редукторами?

Редукторы всегда возвращают аккумулированное состояние (основанное на всех предыдущих и текущей операциях). Они действуют подобно "редукторам состояния". При каждом вызове редуктора, ему в качестве аргументов передаются состояние и операция. Переданное состояние обновляется (аккумулируется с предыдущим) на основе операции, и возвращается новое состояние. Вы можете "редуцировать" несколько операций и начальное состояние (хранилища), применить эти операции к состоянию для получения результирующего состояния.

### Как сделать AJAX-запрос в Redux?

Для этого можно использовать промежуточное программное обеспечение redux-thunk, позволяющее определять асинхронные операции.

Рассмотрим пример запроса определенного аккаунта с помощью fetch API:

export function fetchAccount(id) {

return dispatch => {

dispatch(setLoadingAccountState()) // показываем индикатор загрузки

fetch(`/account/${id}`, (response) => {

dispatch(doneFetchingAccount()) // скрываем индикатор

if (response.status === 200) {

dispatch(setAccount(response.json())) // обновляем состояние полученными данными

} else {

dispatch(someError)

}

})

}

}

function setAccount(data) {

return { type: 'SET\_ACCOUNT', data }

}

### Обязательно ли хранить состояние всех компонентов в хранилище Redux?

Данные приложения следует хранить в хранилище Redux, а состояние компонентов пользовательского интерфейса в соответствующих компонентах. У создателя Redux Дэна Абрамова по этому поводу есть статья под названием "Следует ли вам использовать Redux?"

### Как рекомендуется получать доступ к хранилищу Redux?

Лучшим способом получить хранилище в компоненте является использование функции connect(), которая создает новый компонент, оборачивающий существующий. Этот паттерн называется компоненты высшего порядка, он является предпочтительным способом расширения функциональности компонента в React. Это позволяет передавать в компонент состояние и "создателей операций" (action creators), в том числе, при обновлении хранилища.

Создадим компонент FilterLink с помощью connect():

import { connect } from 'react-redux'

import { setVisibilityFilter } from '../actions'

import Link from '../components/Link'

const mapStateToProps = (state, ownProps) => ({

active: ownProps.filter === state.visibilityFilter

})

const mapDispatchToProps = (dispatch, ownProps) => ({

onClick: () => dispatch(setVisibilityFilter(ownProps.filter))

})

const FilterLink = connect(

mapStateToProps,

mapDispatchToProps

)(Link)

export default FilterLink

Поскольку такой вариант имеет несколько оптимизаций производительности и, как правило, меньше подвержен "багам", разработчики Redux почти всегда рекомендуют использовать connect() вместо прямого доступа к хранилищу (с помощью API контекста).

class MyComponent {

someMethod() {

doSomethingWith(this.context.store)

}

}

### В чем разница между компонентом и контейнером в React Redux?

Component - это классовый или функциональный компонент, описывающий визуальное представление приложения.

Container - это неофициальный термин для описания компонента, подключенного к хранилищу Redux. Контейнеры "подписываются" на обновление состояния Redux и "запускают" (dispatch) операции, они, как правило, не рендерят DOM-элементы: они делегируют рендеринг дочерним компонентам, отвечающим за визуализацию.

### Для чего в Redux нужны константы?

Константы позволяют легко обнаруживать все случаи их применения в проекте при использовании IDE. Они также позволяют избегать глупых ошибок, связанных с типами - немедленно выбрасывается исключение ReferenceError.

Обычно, мы сохраняем константы в отдельном файле (constants.js или actionTypes.js).

export const ADD\_TODO = 'ADD\_TODO'

export const DELETE\_TODO = 'DELETE\_TODO'

export const EDIT\_TODO = 'EDIT\_TODO'

export const COMPLETE\_TODO = 'COMPLETE\_TODO'

export const COMPLETE\_ALL = 'COMPLETE\_ALL'

export const CLEAR\_COMPLETED = 'CLEAR\_COMPLETED'

В Redux мы используем их в двух местах:

* 1. **Во время создания операции:**

actions.js:

import { ADD\_TODO } from './actionTypes';

export function addTodo(text) {

return { type: ADD\_TODO, text }

}

* 1. **В редукторах:**

reducer.js:

import { ADD\_TODO } from './actionTypes'

export default (state = [], action) => {

switch (action.type) {

case ADD\_TODO:

return [

...state,

{

text: action.text,

completed: false

}

];

default:

return state

}

}

### Какие способы существуют для написания mapDispatchToProps()?

Существует несколько способов привязать "создателей операций" к методу dispatch() в mapDispatchToProps().

Ниже представлены возможные варианты:

const mapDispatchToProps = (dispatch) => ({

action: () => dispatch(action())

})

const mapDispatchToProps = (dispatch) => ({

action: bindActionCreators(action, dispatch)

})

const mapDispatchToProps = { action }

Третий вариант является сокращением первого.

### Для чего используется параметр ownProps в методах mapStateToProps() и mapDispatchToProps()?

При определении параметра ownProps React Redux передает пропы в компонент в функциях "подключения". Поэтому, если вы используете подключенный компонент:

import ConnectedComponent from './containers/ConnectedComponent';

<ConnectedComponent user='Иван' />

ownProps внутри функций mapStateToProps() и mapDispatchToProps() будет объектом:

{ user: 'Иван' }

Вы можете использовать этот объект для определения значения, возвращаемого указанными функциями.

### Как структурировать директории верхнего уровня в Redux?

Большинство приложений имеют несколько "топовых" директорий:

* 1. **components**: используется для "тупых" компонентов, не знающих о Redux
  2. **containers**: используется для "умных" компонентов, подключенных к Redux
  3. **actions**: используется для всех создателей операций - названия файлов указывают на соответствующие части приложения
  4. **reducers**: используется для всех редукторов - названия коррелируют с ключами состояния
  5. **store**: используется для инициализации хранилища

Такая структура прекрасно подходит для небольших и средних приложений.

### Что такое redux-saga?

redux-saga - это библиотека, позволяющая легче и быстрее выполнять побочные эффекты (асинхронную логику, вроде получения данных и доступа к кэшу браузера) в React/Redux-приложениях.

Она доступна в NPM:

$ yarn add redux-saga

// или

$ npm i redux-saga

### Определите ментальную модель redux-saga

Saga - своего рода отдельный поток (выполнения кода) в приложении, отвечающий исключительно за побочные эффекты. redux-saga - это middleware для Redux, это означает, что данный "тред" может запускаться, приостанавливаться и отменяться из основного приложения с помощью обычных операций Redux. Он имеет доступ к состоянию приложения и может инициировать запуск операций.

### В чем разница между методами call() и put() в redux-saga?

call() и put() являются функциями создания эффектов. call() используется для создания эффекта описания, указывающего middleware вызвать промис. put() создает эффект, указывающий middleware запустить операцию.

Рассмотрим, как эти эффекты работают применительно к запросу пользовательских данных:

function\* fetchUserSaga(action) {

// функция `call()` получает аргументы, которые передаются функции `api.fetchUser()`,

// указываем middleware вызвать промис, после чего разрешенное значение присваивается переменной userData

const userData = yield call(api.fetchUser, action.userId)

// указываем middleware запустить соответствующую операцию

yield put({

type: 'FETCH\_USER\_SUCCESS',

userData

})

}

### Что такое Redux Thunk?

Redux Thunk - это промежуточное программное обеспечение, которое позволяет писать "создателей операций", возвращающих функции вместо операций. Thunk может использоваться для отложенного или условного запуска операции. Внутренняя функция в качестве параметров принимает методы хранилища dispatch() и getState().

### В чем разница между redux-saga и redux-thunk?

Redux Thunk и Redux Saga предназначены для работы с побочными эффектами. В большинстве сценариев, Thunk для этого пользуется промисами, а Saga - генераторами. Thunk проще в использовании и промисы лучше знакомы большинству разработчиков, Saga предоставляет больше возможностей, но требуется хорошо разбираться в генераторах. Названные middlewares могут использоваться совместно: можно начать с использования Thunk и, при необходимости, перейти на Saga.

### Что такое Redux DevTools?

Redux DevTools - это Redux-окружение для "путешествий во времени" и "живого" редактирования кода с возможностью "горячей" перезагрузки, повторения операций и "кастомизируемым" интерфейсом. Если вы не хотите возиться с установкой Redux DevTools и его интеграцией в свой проект, присмотритесь к соответствующим расширениям для Chrome и Firefox.

### Назовите основные возможности Redux DevTools

Вот некоторые из основных возможностей Redux DevTools:

* 1. Позволяют инспектировать каждое состояние и полезную нагрузку операции
  2. Позволяют возвращаться назад, "отменяя" выполнение операций
  3. При изменении кода редуктора осуществляется повторное вычисление каждой "зафиксированной" операции
  4. Если редуктор выбросил исключение, вы сможете увидеть, в процессе выполнения какой операции это произошло, и в чем заключается ошибка
  5. С помощью метода persistState() можно сохранить сессию отладки между перезагрузками страницы

### Что такое селекторы (selectors) Redux и зачем их использовать?

Selectors - это функции, принимающие состояние Redux в качестве аргумента и возвращающие некоторые данные для передачи компоненту.

Например, так можно извлечь данные пользователя из состояния:

const getUserData = state => state.user.data

Селекторы имеют два главных преимущества:

* 1. Селектор может вычислять производные данные, позволяя Redux записывать в хранилище минимально возможное состояние
  2. Селектор не выполняет повторных вычислений до тех пор, пока не изменится один из его аргументов

### Что такое Redux Form?

Redux Form работает с React и Redux, позволяя формам в React хранить состояние в Redux. Redux Form может использоваться с обычными HTML5-инпутами, а также с популярными UI-фреймворками, такими как Material UI, React Widgets и React Bootstrap.

### Назовите основные возможности, предоставляемые Redux Form?

Вот некоторые из основных особенностей Redux Form:

* 1. Значения полей записываются в хранилище Redux
  2. Синхронная/асинхронная валидация полей и отправка формы
  3. Форматирование, разбор и нормализация значений полей

### Как добавить несколько middleware в Redux?

Для этого можно использовать метод applyMiddleware().

Например, можно добавить redux-thunk и logger, передав их в качестве аргументов в applyMiddleware():

import { createStore, applyMiddleware } from 'redux'

const createStoreWithMiddleware = applyMiddleware(thunk, logger)(createStore)

### Как установить начальное значение в Redux?

Для этого необходимо передать начальное состояние как второй аргумент в метод createStore():

const rootReducer = combineReducers({

todos: todos,

visibilityFilter: visibilityFilter

})

const initialState = {

todos: [{ id: 123, name: 'пример', completed: false }]

}

const store = createStore(

rootReducer,

initialState

)

### Чем Relay отличается от Redux?

Relay похож на Redux тем, что оба используют единственное хранилище. Основное отличие состоит в том, что Relay управляет состоянием через сервер, доступ к состоянию (чтение данных) и его изменение осуществляется через GraphQL-запросы. Relay кэширует данные в целях оптимизации их получения, запрашивая только изменившиеся данные и ничего более.

### Что такое операция (action) в Redux?

Actions - это обычные JavaScript-объекты, содержащие данные приложения, которые отправляются в хранилище. Операции должны иметь свойство type, указывающее какой тип операции необходимо выполнить. Операции также могут содержать полезную нагрузку (payload) - данные для обновления состояния.

Вот как может выглядеть операция по добавлению новой задачи в список:

// здесь используется константа

{

type: ADD\_TODO,

text: 'Добавление задачи в список'

}

## Библиотеки для React

### Что такое reselect и как он работает?

Reselect - это библиотека селекторов (для Redux), которая использует концепцию мемоизации. Изначально она создавалась для вычисления производных данных из состояния Redux-подобных приложений, но может применяться и к другой архитектуре или библиотеке.

Reselect сохраняет копию входных/выходных данных последнего вызова и производит повторные вычисления только при изменении этих данных. Если передаются одни и те же данные, reselect возвращает результат из кэша. Мемоизация и кэширование являются полностью настраиваемыми.

### Что такое Flow?

Flow - это инструмент для статической проверки типов, специально разработанный для обнаружения ошибок в типах JavaScript. Типы Flow делают более мелкие различия, чем традиционная система типов. Например, Flow, в отличие от большинства других систем типов, помогает перехватывать ошибки, связанные с null.

### В чем разница между Flow и PropTypes?

Flow - это инструмент статического анализа, который используется надмножество языка, позволяющее добавлять аннотации типов для всего кода и осуществлять проверку во время компиляции.

PropTypes - это инструмент базовой проверки типов, встроенный в React. Он может проверять только типы пропов, переданных компоненту. Если вам нужна большая гибкость с точки зрения проверки типов, то лучшим выбором будет использование Flow/TypeScript.

### Как использовать иконки font-awesome в React?

Для того, чтобы включить Font Awesome в React-проект, необходимо выполнить следующие шаги:

* 1. Установить font-awesome:

$ yarn add font-awesome

// или

$ npm i font-awesome

* 1. Импортировать font-awesome в файл index.js:

import 'font-awesome/css/font-awesome.min.css'

* 1. Добавить класс Font Awesome в атрибут className:

render() {

return <div><i className={'fa fa-spinner'} /></div>

}

### Что такое React DevTools?

React Developer Tools позволяют инспектировать иерархию компонентов, включая их состояние и пропы. Они существуют как в виде расширения для браузера (Chrome и Firefox), так и в виде самостоятельного приложения (работают с другими окружениями, такими как Safari, IE и React Native).

### Почему DevTools не загружаются в Chrome для локальных файлов?

Если вы открыли локальный HTML-файл в браузере (file://...), тогда вам необходимо открыть Расширения Chrome и выбрать Allow access to file URLs (разрешить доступ к путям файлов).

### Как использовать Polymer в React?

Для того, чтобы использовать Polymer в React, необходимо сделать следующее:

* 1. Создать элемент Polymer:

<link rel='import' href='../../bower\_components/polymer/polymer.html' />

Polymer({

is: 'calender-element',

ready: function() {

this.textContent = 'Это календарь'

}

})

* 1. Создать компонент Polymer, импортировав его в index.html:

<link rel='import' href='./src/polymer-components/calender-element.html'>

* 1. Использовать этот компонент в JSX:

import React from 'react'

class MyComponent extends React.Component {

render() {

return (

<calender-element />

)

}

}

export default MyComponent

### В чем заключаются преимущества React перед Vue?

React имеет следующие преимущества перед Vue:

* 1. Предоставляет большую гибкость при разработке больших приложений
  2. Его легче тестировать
  3. Подходит для разработки мобильных приложений
  4. Доступно больше информации и готовых решений

**Обратите внимание:** приведенные выше преимущества являются субъективными и сильно зависят от опыта разработки.

### В чем разница между React и Angular?

Отличия между React и Angular в табличной форме:

| React | Angular |
| --- | --- |
| React - это библиотека, которая имеет только слой представления | Angular - это фреймворк, в котором реализован весь функционал MVC |
| React выполняет рендеринг на стороне сервера | Angular раньше осуществлял рендеринг на стороне клиента, но, начиная с Angular 2, он также делегировал эти полномочия серверу |
| React использует JSX, который выглядит как HTML в JS, что может сбивать с толку | Angular следует "шаблонному" подходу к HTML, что делает код более коротким и понятным |
| React Native, позволяющий разрабатывать мобильные приложения быстрее и стабильнее | Ionic, соответственно, менее стабильный и более медленный |
| В React поток данных является однонаправленным, что существенно облегчает отладку | В Angular поток данных двунаправленный, данные связывают дочерний и родительский компоненты, что часто затрудняет отладку |

**Обратите внимание:** приведенные отличия является субъективными и сильно зависят от профессионального опыта.

### Почему в DevTools не отображается вкладка React?

При загрузке страницы React DevTools устанавливает глобальную переменную \_\_REACT\_DEVTOOLS\_GLOBAL\_HOOK\_\_, после чего React активирует этот хук в процессе инициализации и использует его для взаимодействия с инструментами разработчика. Если сайт не использует React или React больше не может взаимодействовать с DevTools, соответствующая вкладка не будет отображаться.

### Что такое Styled Components?

styled-components - это JavaScript-библиотека для стилизации React-приложений. Она исключает необходимость интеграции между стилями и компонентами и позволяет писать стили, дополняемые JavaScript.

### Приведите пример использования Styled Components

Создадим компоненты Title и Wrapper с определенными стилями для каждого:

import React from 'react'

import styled from 'styled-components'

const Title = styled.h1`

font-size: 1.5em;

text-align: center;

color: palevioletred;

`

const Wrapper = styled.section`

padding: 4em;

background: papayawhip;

`

Переменные Title и Wrapper являются компонентами, которые можно рендерить как любые другие компоненты:

<Wrapper>

<Title>{'Давайте создадим наш первый стилизованный компонент!'}</Title>

</Wrapper>

### Что такое Relay?

Relay - это фреймворк JavaScript, предоставляющий слой данных и возможность клиент-серверной коммуникации для веб-приложений, в которых используется слой представления React.

### Как создать TypeScript-проект с помощью create-react-app?

Начиная с react-scripts@2.1.0, в create-react-app встроена возможность автоматического создания React/TypeScript-проектов. Для этого достаточно указать --template typescript или просто --typescript после названия приложения:

npx create-react-app my-app --template typescript

# или

yarn create react-app my-app --typescript

## Разное

### Назовите основные возможности библиотеки Reselect?

Вот основные возможности этой библиотеки:

* 1. Селекторы могут вычислять производные данные, позволяя Redux сохранять минимально возможное состояние
  2. Селекторы являются эффективными. Они не выполняют повторных вычислений до тех пор, пока не изменится один из переданных им аргументов
  3. Допустима композиция селекторов. Они могут передаваться другим селекторам

### Приведите пример использования Reselect

Выполним некоторые вычисления, связанные с заказом товара, с помощью Reselect:

import { createSelector } from 'reselect'

const shopItemsSelector = state => state.shop.items

const taxPercentSelector = state => state.shop.taxPercent

const subtotalSelector = createSelector(

shopItemsSelector,

items => items.reduce((acc, item) => acc + item.value, 0)

)

const taxSelector = createSelector(

subtotalSelector,

taxPercentSelector,

(subtotal, taxPercent) => subtotal \* (taxPercent / 100)

)

export const totalSelector = createSelector(

subtotalSelector,

taxSelector,

(subtotal, tax) => ({ total: subtotal + tax })

)

const exampleState = {

shop: {

taxPercent: 8,

items: [

{ name: 'яблоко', value: 1.20 },

{ name: 'апельсин', value: 0.95 },

]

}

}

console.log(subtotalSelector(exampleState)) // 2.15

console.log(taxSelector(exampleState)) // 0.172

console.log(totalSelector(exampleState)) // { total: 2.322 }

### Можно ли использовать статические объекты в классовых компонентах React?

Нет, statics работает только в React.createClass():

someComponent= React.createClass({

statics: {

someMethod: function() {

// ...

}

}

})

Однако, вы можете создавать статические элементы внутри классов с помощью синтаксиса статических полей класса:

class Component extends React.Component {

static propTypes = {

// ...

}

static someMethod() {

// ...

}

}

Или снаружи класса:

class Component extends React.Component {

// ...

}

Component.propTypes = {...}

Component.someMethod = function(){....}

### Redux может использоваться только с React?

Redux может использоваться в качестве хранилища данных для любого пользовательского интерфейса. Чаще всего, он используется вместе с React и React Native, но существует возможность его интеграции с Angular, Angular 2, Vue, Mithril и т.д. Redux просто предоставляет механизм подписки, который может использоваться в любом коде.

### Требуются ли какие-либо дополнительные инструменты для работы с Redux?

Redux написан на синтаксисе ES6 и транспилируется в ES5 при сборке для продакшна с помощью Webpack и Babel. Вам не требуются какие-либо дополнительные инструменты. Redux также предоставляется в виде UMD-модуля, что позволяет использовать его напрямую, минуя стадию сборки.

### Как обновить Redux Form initialValues с помощью состояния?

Для этого необходимо добавить настройку enableReinitialize : true.

const InitializeFromStateForm = reduxForm({

form: 'initializeFromState',

enableReinitialize : true

})(UserEdit)

**Обратите внимание:** обновление начальных значений повлечет за собой обновление формы.

### Как с помощью React PropTypes разрешить использование разных типов одним пропом?

Для этого можно использовать метод oneOfType().

Например, значение свойства height может быть указано с помощью типа string или number:

Component.PropTypes = {

height: PropTypes.oneOfType([

PropTypes.string,

PropTypes.number

])

}

### Можно ли импортировать SVG-файл как компонент React?

Вы можете импортировать SVG в компонент вместо его загрузки в виде файла. Эта возможность доступна, начиная с react-scripts@2.0.0.

import { ReactComponent as Logo } from './logo.svg'

const App = () => (

<div>

{/\* Logo - это обычный React-компонент \*/}

<Logo />

</div>

)

**Обратите внимание**: не забудьте использовать фигурные скобки при импорте

### Почему не рекомендуется использовать встроенные "реф-коллбеки" или функции?

Если коллбек ссылки определен как встроенная функция, он будет вызван дважды в процессе обновления, в первый раз с нулевым значением, второй - с DOM-элементом. Это объясняется тем, что новый экземпляр функции создается при каждом рендеринге, поэтому React необходимо удалить старую ссылку и установить новую:

class UserForm extends Component {

handleSubmit = () => {

console.log("Значением поля является: ", this.input.value)

}

render () {

return (

<form onSubmit={this.handleSubmit}>

<input

type='text'

ref={(input) => this.input = input} /> // это позволяет получить доступ к значению input в обработчике submit

<button type='submit'>Отправить</button>

</form>

)

}

}

Однако, нам нужно, чтобы коллбек вызывался один раз при монтировании компонента. Быстрым решением является использование синтаксиса полей класса для определения функции:

class UserForm extends Component {

handleSubmit = () => {

console.log("Значением поля является: ", this.input.value)

}

setSearchInput = (input) => {

this.input = input

}

render () {

return (

<form onSubmit={this.handleSubmit}>

<input

type='text'

ref={this.setSearchInput} />

<button type='submit'>Отправить</button>

</form>

)

}

}

**Обратите внимание:** при использовании хука useRef() в функциональных компонентах таких проблем не возникает.

### Что такое Render Hijacking в React?

Концепция перехвата рендеринга - это возможность контролировать, что получит один компонент от другого. Это означает, что вы декорируете компонент, оборачивая его в HOC. Это позволяет внедрять дополнительные пропы или осуществлять другие изменения, которые меняют логику рендеринга. В действительности, не происходит никакого перехвата, но использование HOC заставляет компонент вести себя по-другому.

### Как реализовать HOC-фабрику?

Для реализации HOC в React существует два способа.

* 1. Проксирование пропов (Props Proxy - PP) и
  2. Инверсия наследования (Inheritance Inversion - II).

Они используют разные подходы к управлению WrappedComponent (обернутым компонентом).

**Проксирование пропов**

При использовании данного подхода метод render() возвращает React-элемент с типом WrappedComponent. Мы также передаем пропы, полученные HOC, поэтому данный подход называется Props Proxy:

function ppHOC(WrappedComponent) {

return class PP extends React.Component {

render() {

return <WrappedComponent {...this.props}/>

}

}

}

**Инверсия наследования**

При использовании данного подхода возвращаемый HOC класс (Enhancer - усилитель) расширяет WrappedComponent. Он называется Inheritance Inversion потому, что вместо того, чтобы WrappedComponent расширял некоторый класс Enhancer, он сам пассивно расширяется "усилителем". Отношения между ними напоминают инверсию.

function iiHOC(WrappedComponent) {

return class Enhancer extends WrappedComponent {

render() {

return super.render()

}

}

}

### Как передать число в React-компонент?

Числа просто заключаются в фигурные скобки ({}), а строки дополнительно закавычиваются:

React.render(<User age={30} department={"IT"} />, document.getElementById('container'));

### Обязательно ли хранить все состояние в Redux? Можно ли использовать внутреннее состояние компонентов?

Вы сами принимаете решение, что использовать. В этом заключается работа разработчика - определить, какое состояние требуется приложению и где должна храниться каждая часть этого состояния. Одни разработчики предпочитают хранить все состояние в Redux, что обеспечивает полную сериализацию и управляемость приложения. Другие предпочитают хранить некритичное состояние UI, такое как "открыт ли выпадающий список" внутри компонента.

Ниже представлены основные правила определения того, какие типы данных следует хранить в Redux:

* 1. Нуждаются ли другие части приложения в этих данных?
  2. Требуется ли создавать производные данные на основе оригинальных?
  3. Используются ли эти данные несколькими компонентами?
  4. Существует ли вероятность того, что потребуется восстанавливать прошлое состояние?
  5. Собираетесь ли вы кэшировать данные (для использования версии из кэша вместо повторного запроса)?

### Для чего предназначен метод registerServiceWorker() в React?

React создает сервис-воркера без настройки по умолчанию. Сервис-воркер - это веб-API, позволяющее записывать файлы приложения в кэш и возвращать их из него при отсутствии подключения к сети или медленном соединении, что сильно улучшает пользовательский опыт. Сервис-воркер - это своего рода прокси для HTTP-запросов.

import React from 'react';

import ReactDOM from 'react-dom';

import App from './App';

import registerServiceWorker from './registerServiceWorker';

ReactDOM.render(<App />, document.getElementById('root'));

registerServiceWorker();

### Что такое React.memo()?

Классовым компонентам можно запретить повторный рендеринг, если их пропы остались прежними, с помощью PureComponent или shouldComponentUpdate(). Теперь и у функциональных компонентов имеется такая возможность благодаря функции-обертке React.memo():

const MyComponent = React.memo(function MyComponent(props) {

/\* повторный рендеринг выполняется только при изменении пропов \*/

});

Похожий функционал предоставляет хук useMemo().

### Что такое React.lazy()?

Функция React.lazy() позволяет рендерить результаты динамического импорта как обычные компоненты. Она автоматически загружает "бандл", содержащий OtherComponent (см. пример ниже), когда данный компонент будет отрендерен. Функция возвращает промис, который разрешается модулем с экспортом по умолчанию, содержащим React-компонент. OtherComponent должен быть обернут в компонент Suspense:

import React, { Suspense } from 'react

const OtherComponent = React.lazy(() => import('./OtherComponent'));

function MyComponent() {

return (

<Suspense>

<OtherComponent />

</Suspense>

);

}

**Обратите внимание:** React.lazy() и Suspense пока не доступны для рендеринга на стороне сервера. Если вам требуется разделение кода в приложении, которое рендерится на сервере, используйте библиотеку React Loadable.

### Как предотвратить лишние обновления с помощью setState()?

Вы можете провести сравнение текущего состояния с существующим и решить, следует повторно рендерить страницу или нет. Если значения одинаковые, для предотвращения повторного рендеринга возвращаем null, иначе, возвращаем последнее значение состояния.

Вот пример условного рендеринга профиля пользователя:

getUserProfile = user => {

const latestAddress = user.address;

this.setState(state => {

if (state.address === latestAddress) {

return null;

} else {

return { title: latestAddress };

}

});

};

### Как рендерить числа, строки и массивы в React 16?

**Массивы**: в отличие от предыдущих версий, метод render() не обязательно должен возвращать единственный элемент. Он вполне может возвращать несколько дочерних элементов без обертки.

Например, вот список разработчиков:

const ReactJSDevs = () => {

return [

<li key="1">John</li>,

<li key="2">Jane</li>

];

}

Вы можете объединить этот массив элементов с другим компонентом:

const JSDevs = () => {

return (

<ul>

<li>Bob</li>

<ReactJSDevs/>

<li>Alice</li>

</ul>

);

}

**Строки и числа:** вы также можете возвращать строку или число из метода render():

render() {

return 'Добро пожаловать в список вопросов по React';

}

// число

render() {

return 2021;

}

### Как использовать синтаксис "определения полей классов" в классовых компонентах?

Синтаксис классовых компонентов может быть сильно сокращен с помощью определения полей классов. Вы можете инициализировать локальное состояние без конструктора и определить методы класса с помощью стрелочных функций без их привязки к экземпляру.

Рассмотрим пример счетчика, в котором используется названный синтаксис:

class Counter extends Component {

state = { value: 0 };

handleIncrement = () => {

this.setState(prevState => ({

value: prevState.value + 1

}));

};

handleDecrement = () => {

this.setState(prevState => ({

value: prevState.value - 1

}));

};

render() {

return (

<div>

{this.state.value}

<button onClick={this.handleIncrement}>+</button>

<button onClick={this.handleDecrement}>-</button>

</div>

)

}

}

### Что такое хуки?

Хуки - это относительно новая возможность, представленная в React 16.8, позволяющая использовать состояние и другие "реактивные" возможности без написания классов.

Вот пример использования хука useState():

import { useState } from 'react';

function Example() {

// переменная count содержит значение состояния компонента

// setCount - функция для обновления этого значения

// вы можете думать об этом, как о паре геттер/сеттер

const [count, setCount] = useState(0);

return (

<div>

<p>Вы кликнули {count} раз</p>

<button onClick={() => setCount(count + 1)}>

Нажми на меня

</button>

</div>

);

}

### Назовите правила использования хуков?

При использовании хуков необходимо соблюдать два правила:

* 1. Хуки не должны вызываться внутри циклов, условий или вложенных функций. Это позволяет обеспечить одинаковый порядок вызова хуков при повторном рендеринге и сохранять состояние хуков между несколькими вызовами useState() и useEffect()
  2. Хуки можно вызывать только внутри функциональных компонентов React и других хуков, вы не должны вызывать их внутри обычных функций

### Как обеспечить соблюдение правил использования хуков?

Команда React разработала специальный ESLint-плагин, который следит за соблюдением правил использования хуков. Вы можете добавить этот плагин в существующий проект, выполнив команду **eslint-plugin-react-hooks**:

yarn add eslint-plugin-react-hooks@next

// или

npm i eslint-plugin-react-hooks@next

И добавив в настройки ESLint следующее:

// настройки линтера

{

"plugins": [

// ...

"react-hooks"

],

"rules": {

// ...

"react-hooks/rules-of-hooks": "error"

}

}

**Обратите внимание:** данный плагин применяется по умолчанию при использовании Create React App для создания проекта.

### В чем разница между Flux и Redux?

Ниже представлены основные отличия между Flux и Redux:

| Flux | Redux |
| --- | --- |
| Состояние мутабельно | Состояние иммутабельно |
| Хранилище содержит как состояние, так и логику его изменения | Хранилище содержит только состояние |
| Существует несколько хранилищ | Существует только одно хранилище |
| Все хранилища являются самостоятельными и "плоскими" | Одно хранилище, содержащее иерархию редукторов |
| Имеется диспетчер-одиночка ("синглтон") | Концепция диспетчера как таковая отсутствует |
| React-компоненты подписываются на хранилище | Компоненты-контейнеры используют функцию connect() |

### В чем заключаются преимущества использования React Router 4?

Ниже представлены основные преимущества использования новой версии React Router:

* 1. API реализует компонентный подход. Роутер представлен в виде обычного компонента (BrowserRouter, например), который оборачивает дочерние компоненты (Route и др.)
  2. Нет необходимости работать с историей напрямую. Роутер сам об этом позаботится - главное, не забудьте обернуть маршруты в Router
  3. Размер приложения уменьшается за счет использования только определенных модулей (модули ядра, веб или нативные модули)

### Опишите сигнатуру метода жизненного цикла componentDidCatch()

Метод componentDidCatch() вызывается после того, как в любом из его потомков выбрасывается исключение. Метод принимает два параметра:

* 1. error - объект выброшенного исключения
  2. info - объект с ключом componentStack, содержащим информацию о том, какой компонент выбросил исключение

Структура метода:

componentDidCatch(error, info)

### В каких случаях предохранители не перехватывают ошибки?

Предохранители не срабатывают в следующих случаях:

* 1. Внутри обработчиков событий
  2. В асинхронном коде, использующем коллбеки setTimeout() или requestAnimationFrame()
  3. В процессе рендеринга на стороне сервера
  4. Когда исключение выбрасывается в самом предохранителе

### Почему в обработчиках событий предохранители не нужны?

Предохранители не перехватывают ошибки внутри обработчиков событий. Обработчики не вызываются во время рендеринга, в отличие от метода render() или методов жизненного цикла. Поэтому React знает, как справиться с ошибками в обработчиках. Если вам все же требуется "перехватчик" ошибок за пределами обработчика, используйте блок try/catch.

class MyComponent extends React.Component {

constructor(props) {

super(props);

this.state = { error: null };

}

handleClick = () => {

try {

// ...

} catch (error) {

this.setState({ error });

}

}

render() {

if (this.state.error) {

return <h1>Caught an error.</h1>

}

return <div onClick={this.handleClick}>Click Me</div>

}

}

В примере ошибка перехватывается с помощью try/catch вместо предохранителя.

### В чем разница между блоком try/catch и предохранителями?

try/catch предназначен для работы с императивным кодом, а предохранители - с декларативным, результаты которого отображаются на экране.

Пример использования try/catch:

try {

showButton();

} catch (error) {

// ...

}

Пример использования предохранителя:

<ErrorBoundary>

<MyComponent />

</ErrorBoundary>

Если где-то глубоко в дереве компонента возникнет ошибка, она будет обработана ближайшим предохранителем.

### Как обрабатываются неперехваченные ошибки в React 16?

В React 16 ошибки, которые не были перехвачены предохранителем, приведут к размонтированию всего дерева компонента. Причина такого решения состоит в том, что лучше полностью удалить интерфейс, чем отобразить его неправильную версию. Например, лучше ничего не отобразить, чем отобразить неправильную сумму в приложении для оплаты.

### Куда следует помещать предохранители?

Это зависит от потребностей приложения. Вы можете использовать один из следующих подходов:

* 1. Обернуть в предохранитель маршрутизаторы верхнего уровня для отображения общего сообщения об ошибке для всего приложения
  2. Либо оборачивать отдельные компоненты во избежание "падения" всего приложения

### В чем заключается преимущество трассировки стека компонента перед предохранителями?

Кроме сообщений об ошибках и обычной трассировки стека, React отображает трассировку стека компонента с названиями компонентов и пронумерованными строками с помощью концепции предохранителей.

### Какой метод является обязательным для классового компонента?

Единственный обязательным методом классового компонента React является метод render(), остальные методы являются опциональными.

### Какие типы может возвращать метод render()?

Метод render() может возвращать следующие типы:

* 1. **React-элементы:** элементы, преобразуемые в узлы DOM. Они включают в себя HTML-элементы, такие как <div> и пользовательские элементы
  2. **Массивы и фрагменты:** элементы, представляющие собой обертку для нескольких дочерних элементов
  3. **Порталы:** позволяют рендерить дочерние элементы в другом поддереве DOM
  4. **Строки и числа:** встраиваются в DOM в виде текстовых узлов
  5. **Логические значения и null:** не отображаются на экране, используются для условного рендеринга
  6. **Некоторые другие**

### В чем заключается основное назначение конструктора?

Конструктор предназначен для:

* 1. Инициализации локального состояния посредством присваивания this.state какого-либо объекта
  2. Для привязки методов обработчиков событий к экземпляру

Пример обоих случаев:

constructor(props) {

super(props);

// вызывать this.setState() здесь нельзя!

this.state = { counter: 0 };

this.handleClick = this.handleClick.bind(this);

}

### Обязательно ли определять конструктор в компоненте React?

Нет, это не является обязательным. Если вам не нужно инициализировать состояние или определять контекст методов, тогда можно реализовать компонент без конструктора. Инициализировать состояние экземпляра также можно с помощью синтаксиса определения полей класса.

### Что такое пропы по умолчанию (default props)?

Свойство defaultProps определяет пропы, которые устанавливаются классу по умолчанию. Значения данного объекта используются для неопределенных или "нулевых" пропов.

Создадим дефолтный проп для цвета компонента кнопки:

class MyButton extends React.Component {

// ...

}

MyButton.defaultProps = {

color: 'red'

};

Если при использовании компонента MyButton ему не будет передан проп color, значением этого пропа станет значение по умолчанию, т.е. red.

render() {

return <MyButton /> ; // значением props.color будет red

}

**Обратите внимание:** если вы передадите null, значением пропа будет null.

### Почему не следует вызывать setState() в componentWillUnmount()?

setState() не следует вызывать в componentWillUnmount(), поскольку после размонтирования компонент больше не монтируется, а, значит, состояние компонента никогда не обновится.

### Для чего используется getDerivedStateFromError()?

Данный метод вызывается после того, как один из дочерних компонентов выбросил исключение. Он получает ошибку в качестве аргумента и должен вернуть значение для обновления состояния.

Сигнатура этого метода выглядит так:

static getDerivedStateFromError(error)

Рассмотрим случай использования предохранителя:

class ErrorBoundary extends React.Component {

constructor(props) {

super(props);

this.state = { hasError: false };

}

static getDerivedStateFromError(error) {

// обновляем состояние для отображения запасного контента при следующем рендеринге

return { hasError: true };

}

render() {

if (this.state.hasError) {

// вы можете рендерить любой резервный интерфейс

return <h1>Что-то пошло не так.</h1>;

}

return this.props.children;

}

}

### В каком порядке вызываются методы при повторном рендеринге компонента?

Обновление может быть вызвано изменением пропов или состояния. При повторном рендеринге методы вызываются в следующим порядке:

* 1. static getDerivedStateFromProps()
  2. shouldComponentUpdate()
  3. render()
  4. getSnapshotBeforeUpdate()
  5. componentDidUpdate()

### Какие методы вызываются при обработке ошибок?

При возникновении ошибки в процессе рендеринга, в методе жизненного цикла, в конструкторе или любом потомке, вызываются следующие методы:

* 1. static getDerivedStateFromError()
  2. componentDidCatch()

### Для чего используется поле класса displayName?

displayName используется для отладки. Обычно, вам не нужно определять его явно, оно ссылается на название функции или класса, определяющего компонент. Явное обозначение может потребоваться для отображения другого названия в целях отладки или при создании компонента высшего порядка.

В следующем примере displayName используется для указания на то, что мы имеем дело с результатом HOC withSubscription:

function withSubscription(WrappedComponent) {

class WithSubscription extends React.Component {/\* ... \*/}

WithSubscription.displayName = `WithSubscription(${getDisplayName(WrappedComponent)})`;

return WithSubscription;

}

function getDisplayName(WrappedComponent) {

return WrappedComponent.displayName || WrappedComponent.name || 'Component';

}

### Хорошо ли React-приложения поддерживаются браузерами?

React поддерживается всеми популярными браузерами, включая Internet Explorer 9 и выше. Для более старых браузеров требуются полифилы.

### Для чего используется метод жизненного цикла unmountComponentAtNode()?

Данный метод из пакета react-dom используется для удаления смонтированного React-компонента из DOM, очистки его обработчиков и состояния. Если целевой компонент отсутствует, этот метод ничего не делает. Возвращает true при размонтировании компонента и false в противном случае.

Названный метод имеет следующую сигнатуру:

ReactDOM.unmountComponentAtNode(container)

### Что такое разделение кода (code splitting)?

Разделение кода - это техника, используемая сборщиками модулей, такими как Webpack и Browserify, когда создается несколько "бандлов", подгружаемых по необходимости. React поддерживает разделение кода с помощью динамического импорта.

В приведенном ниже сниппете moduleA.js выделяется в отдельный "чанк" (chunk - часть, кусок), который загружается только после того, как пользователь нажал кнопку "Загрузить":

**moduleA.js**

const moduleA = 'Привет';

export { moduleA };

**App.js**

import React, { Component } from 'react';

class App extends Component {

handleClick = () => {

import('./moduleA')

.then(({ moduleA }) => {

// использование модуля

})

.catch(err => {

// обработка провала

});

};

render() {

return (

<div>

<button onClick={this.handleClick}>Загрузить</button>

</div>

);

}

}

export default App;

Данная техника также используется для отложенной ("ленивой") загрузки модулей с помощью функции React.lazy() и компонента Suspense.

### В чем заключаются преимущества использования строгого режима?

StrictMode может быть полезен в следующих случаях:

* 1. Идентификация компонентов с небезопасными методами жизненного цикла
  2. Вывод предупреждений об использовании устаревших строковых ссылок
  3. Определение неожиданных побочных эффектов
  4. Определение использования устаревшего API контекста
  5. Вывод предупреждений об использовании устаревшего метода findDOMNode()

### Что такое фрагменты с ключами?

Фрагменты, определяемые с помощью React.Fragment, могут иметь ключи. Типичным примером такого использования является создание коллекции фрагментов:

function Glossary(props) {

return (

<dl>

{props.items.map(item => (

// без `key` React выведет предупреждение в консоль

<React.Fragment key={item.id}>

<dt>{item.term}</dt>

<dd>{item.description}</dd>

</React.Fragment>

))}

</dl>

);

}

**Обратите внимание:** key - это единственный атрибут, который можно добавлять фрагменту. В будущем появится возможность добавлять другие атрибуты, такие как обработчики событий.

### Все ли HTML-атрибуты поддерживаются React?

Начиная с React 16, полностью поддерживаются все стандартные и пользовательские DOM-атрибуты. Поскольку в React-компонентах используются как "кастомные", так и связанные с DOM пропы, в React используется camelCase, как и в DOM API.

Примеры использования стандартных HTML-атрибутов:

<div tabIndex="-1" /> // node.tabIndex DOM API

<div className="Button" /> // node.className DOM API

<input readOnly={true} /> // node.readOnly DOM API

Данные пропы работают по аналогии с соответствующими HTML-атрибутами, за исключением некоторых особых случаев. Также поддерживаются все SVG-атрибуты.

### Какие ограничения имеют HOC?

Компоненты, кроме очевидных преимуществ, имеют некоторые ограничения:

* 1. **HOC не рекомендуется использовать внутри метода**render()**:**

render() {

// при каждом рендере создается новая версия EnhancedComponent

// EnhancedComponent1 !== EnhancedComponent2

const EnhancedComponent = enhance(MyComponent);

// это приводит к тому, что внутреннее поддерево каждый раз размонтируется/монтируется

return <EnhancedComponent />;

}

Приведенный код ухудшает производительность за счет повторного монтирования компонента, что также приводит к потере состояния компонента и всех его потомков. В данном случае HOC следует переместить за пределы компонента: это приведет к однократному монтированию результирующего компонента.

* 1. **Статические методы не копируются автоматически:**

При применении HOC к компоненту, новый компонент не будет иметь статических методов оригинального компонента:

// определяем статический метод

WrappedComponent.staticMethod = function() {/\*...\*/}

// применяем HOC

const EnhancedComponent = enhance(WrappedComponent);

// "усиленный" компонент не имеет статического метода

typeof EnhancedComponent.staticMethod === 'undefined' // true

Эту проблему можно решать посредством копирования методов в контейнер перед его возращением:

function enhance(WrappedComponent) {

class Enhance extends React.Component {/\*...\*/}

// необходимо точно знать, какие методы копировать

Enhance.staticMethod = WrappedComponent.staticMethod;

return Enhance;

}

* 1. **Рефы не передаются:**

В случае с HOC приходится передавать все пропы оборачиваемому компоненту, но это не работает со ссылками. Это объясняется тем, что ссылка - это не совсем обычный проп, такой как ключ, например. Для передачи рефов следует использовать React.forwardRef().

### Как отлаживать forwardRefs в DevTools?

React.forwardRef() принимает рендер-функцию в качестве параметра и DevTools используют эту функцию для определения того, что следует отображать для компонента, передаваемого по ссылке.

Например, если вы не укажете имя функции рендеринга или не используете свойство displayName, тогда в DevTools функция отобразится как ForwardRef:

const WrappedComponent = React.forwardRef((props, ref) => {

return <LogProps {...props} forwardedRef={ref} />;

});

Если же вы именуете рендер-функцию, тогда она отобразится как ForwardRef(myFunction):

const WrappedComponent = React.forwardRef(

function myFunction(props, ref) {

return <LogProps {...props} forwardedRef={ref} />;

}

);

В качестве альтернативы вы можете установить свойство displayName для функции forwardRef():

function logProps(Component) {

class LogProps extends React.Component {

// ...

}

function forwardRef(props, ref) {

return <LogProps {...props} forwardedRef={ref} />;

}

// определяем отображаемое название компонента

// например, "ForwardRef(logProps(MyComponent))"

const name = Component.displayName || Component.name;

forwardRef.displayName = `logProps(${name})`;

return React.forwardRef(forwardRef);

}

### В каких случаях пропы компонента по умолчанию имеют значение true?

Если вы не передаете значение для пропа, его значением становится true. Такое поведение соответствует поведению HTML.

Приведенные ниже выражения эквиваленты:

<MyInput autocomplete />

<MyInput autocomplete={true} />

**Обратите внимание:** данный подход использовать не рекомендуется, поскольку он может конфликтовать с сокращенной формой записи объектов в JavaScript (например, { name } является сокращением для { name: name }).

### Что таколе NextJS? Назовите его основные возможности

Next.js - это популярный и легковесный фреймворк для статических приложений и приложений с серверным рендерингом, построенных с помощью React. Он также предоставляет решения для стилизации и маршрутизации. Ниже представлены основные возможности данного фреймворка:

* 1. Рендеринг на стороне сервера по умолчанию
  2. Автоматическое разделение кода для ускорения загрузки страниц
  3. Простая клиентская маршрутизация (основанная на страницах (pages))
  4. Среда для разработки, основанная на Webpack, с поддержкой "горячей" перезагрузки модулей (Hot Module Replacement, HMR)
  5. Возможность реализации сервера на Express или любом другом фреймворке для Node.js
  6. Возможность самостоятельной настройки Babel и Webpack

### Как передать обработчик события компоненту?

Вы можете передавать обработчики событий и другие функции дочерним компонентам как пропы. Они затем могут быть использованы следующим образом:

<button onClick={this.handleClick}>

### Является ли использование стрелочных функций в методе render() хорошей практикой?

Да, вы вполне можете их использовать. Часто, самым легким способом является передача аргументов в функцию обратного вызова. Но при таком подходе необходимо внимательно следить за производительностью:

class Foo extends Component {

handleClick() {

console.log('Произошло нажатие на кнопку');

}

render() {

return <button onClick={() => this.handleClick()}>Нажми на меня</button>;

}

}

**Обратите внимание:** использование стрелочной функции в методе render() приводит к созданию функции при каждом рендеринге, что может повлечь проблемы с производительностью.

### Как предотвратить множественный вызов функции?

Если вы используете обработчик событий, такой как onClick() или onScroll(), и хотите предотвратить слишком ранний вызов этого обработчика, тогда вы можете ограничить количество вызовов коллбека. Для этого можно применить одну из следующих техник:

* 1. **Throttling:** за единицу времени можно выполнить только один вызов. Данную технику можно реализовать с помощью функции \_.throttle() библиотеки Lodash
  2. **Debouncing:** вызов будет выполнен по истечении определенного времени. Данную технику можно реализовать с помощью функции \_.debounce()
  3. **RequestAnimationFrame throttling:** вызовы основаны на requestAnimationFrame(). Данную технику можно реализовать с помощью функции \_.raf-schd()

### Как JSX предотвращает атаки, связанные с инъекцией вредоносного кода?

React DOM "обезвреживает" все значение, содержащиеся в JSX, перед их рендерингом. Это исключает возможность внедрения постороннего кода в приложение. Все значение конвертируются в строку перед отрисовкой.

Вот как используются данные, введенные пользователем:

const name = response.potentiallyMaliciousInput;

const element = <h1>{name}</h1>;

Это также позволяет предотвратить межсайтовый скриптинг (XSS).

### Как обновить отрендеренный элемент?

Вы можете обновить UI (представленный отрисованным элементом), передав методу render() ReactDOM новый элемент.

Пример часов, которые обновляются каждую секунду посредством вызова метода рендеринга:

function tick() {

const element = (

<div>

<h1>Привет, народ!</h1>

<h2>Сейчас {new Date().toLocaleTimeString()}.</h2>

</div>

);

ReactDOM.render(element, document.getElementById('root'));

}

setInterval(tick, 1000);

### Почему пропы доступны только для чтения?

Компонент в виде функции или класса никогда не должен модифицировать собственные пропы.

function capital(amount, interest) {

return amount + interest;

}

Приведенная функция называется "чистой", потому что она не изменяет передаваемые ей значения и всегда возвращает одинаковый результат для одних и тех же аргументов. React следует концепции "Все компоненты должны действовать подобно "чистым" функциям по отношению к пропам".

### Почему состояние обновляется путем объединения?

При вызове setState() в компоненте React объединяет переданный объект с текущим состоянием.

Здесь у нас имеется пользователь с состоянием в виде массивов для постов и комментариев:

constructor(props) {

super(props);

this.state = {

posts: [],

comments: []

};

}

Вы можете обновлять эти массивы по отдельности:

componentDidMount() {

fetchPosts().then(response => {

this.setState({

posts: response.posts

});

});

fetchComments().then(response => {

this.setState({

comments: response.comments

});

});

}

В приведенном примере this.setState({ comments }) обновляет только массив с комментариями, не затрагивая массив с постами.

**Обратите внимание:** при использовании хука useState() состояния не объединяются автоматически. При вызове setState() необходимо сливать состояния вручную:

const [state, setState] = useState({

posts: [],

comments: []

})

useEffect(() => {

fetchPosts().then(({ posts }) => {

setState({

...state,

posts

})

})

}, [])

useEffect(() => {

fetchPosts().then(({ comments }) => {

setState({

...state,

comments

})

})

}, [])

### Как передать аргумент в обработчик событий?

При выполнении итераций или циклов обычной практикой является передача дополнительного параметра обработчику событий. Это может быть реализовано с помощью стрелочной функции или метода bind().

Пример обновления таблицы пользователей:

<button onClick={(e) => this.updateUser(userId, e)}>Обновить данные пользователя</button>

<button onClick={this.updateUser.bind(this, userId)}>Обновить данные пользователя</button>

В обоих случаях "синтетический" аргумент передается в качестве второго аргумента. При использовании стрелочных функций, его необходимо передавать явно, при использовании bind(), он передается автоматически.

### Как предотвратить рендеринг компонента?

Это можно сделать, вернув null из компонента. Таким способом можно реализовать условный рендеринг компонента:

function Greeting(props) {

if (!props.loggedIn) {

return null;

}

return (

<div className="greeting">

Добро пожаловать, {props.name}!

</div>

);

}

class User extends React.Component {

constructor(props) {

super(props);

this.state = {loggedIn: false, name: 'Иван'};

}

render() {

return (

<div>

// предотвращаем рендеринг компонента, если не выполнен вход в систему

<Greeting loggedIn={this.state.loggedIn} />

<UserDetails name={this.state.name}>

</div>

);

}

В приведенном примере компонент Greeting не отображается на экране и возвращает нулевое значение.

### Назовите условия для безопасного использования индексов в качестве ключей

Для безопасного использования индексов в качестве ключей требуется соблюдение трех условий:

* 1. Список и его элементы является статическими - они не вычисляются и не изменяются
  2. Элементы списка не имеют идентификаторов
  3. Список не фильтруется, порядок его элементов не меняется

### Должны ли ключи быть уникальными в глобальном контексте?

Ключи должны быть уникальными среди соседних элементов, но не в глобальном контексте. Это означает, что одни и те же ключи можно использовать в разных массивах.

Пример использования одинаковых ключей в разных блоках:

function Book(props) {

const index = (

<ul>

{props.pages.map((page) =>

<li key={page.id}>

{page.title}

</li>

)}

</ul>

);

const content = props.pages.map((page) =>

<div key={page.id}>

<h3>{page.title}</h3>

<p>{page.content}</p>

<p>{page.pageNumber}</p>

</div>

);

return (

<div>

{index}

<hr />

{content}

</div>

);

}

### Назовите популярное решение для обработки форм в React

Formik - это один из наиболее популярных инструментов для работы с формами в React. Эта библиотека предоставляет готовые решения для валидации, отслеживания заполненных полей и реализации отправки формы.

Возможности названной библиотеки можно разделить на следующие категории:

* 1. Запись и получение значений из состояния формы
  2. Валидация и сообщения об ошибках
  3. Обработка отправки формы

Она используется для создания масштабируемых, производительных обработчиков форм минимальными усилиями.

### В чем заключаются преимущества Formik перед Redux Form?

Ниже представлены основные причины, по которым следует предпочесть Formik вместо Redux Form:

* 1. Состояние формы является кратковременным и локальным, так что отсутствует необходимость в его фиксировании с помощью Redux (или любой другой Flux-библиотеки)
  2. Redux Form вызывает "топовый" редуктор при нажатии каждой клавиши. Это может привести к "торможению" ввода в больших приложениях
  3. Размер минифицированной и сжатой Redux Form составляет 22.5 Кб, а Formik всего 12.7 Кб

### Почему вам не требуется работать с наследованием?

В React вместо наследования рекомендуется использовать композицию для обеспечения возможности как совместного использования кода несколькими компонентами, так и повторного использования самих компонентов. Композиция вместе с пропами предоставляют все необходимое для кастомизации внешнего вида и поведения компонента явным и безопасным способом.

Если вам требуется совместное использование функционала, не связанного с UI, тогда следует вынести соответствующий код в отдельный модуль. Компоненты смогут импортировать этот модуль и использовать функцию, объект или класс без необходимости их расширения.

### Можно ли использовать веб-компоненты в React-приложении?

Да, вы вполне можете использовать веб-компоненты в React-приложении. Даже несмотря на то, что многим разработчикам не нравится такое сочетание, это может потребоваться при использовании сторонних библиотек компонентов пользовательского интерфейса, написанных с помощью веб-компонентов.

Пример использования веб-компонента для выбора даты Vaadin:

import React, { Component } from 'react';

import './App.css';

import '@vaadin/vaadin-date-picker';

class App extends Component {

render() {

return (

<div className="App">

<vaadin-date-picker label="Когда вы родились?"></vaadin-date-picker>

</div>

);

}

}

export default App;

### Что такое динамической импорт?

Динамический импорт - это синтаксис, позволяющий реализовать разделение кода, т.е. загрузку модулей по необходимости:

* 1. **Обычный импорт:**

import { add } from './math';

console.log(add(10, 20));

* 1. **Динамический импорт:**

import("./math").then(math => {

console.log(math.add(10, 20));

});

### Что такое загружаемые (loadable) компоненты?

Если вам требуется разделение кода в приложении с серверным рендерингом, рекомендуемым способом является использование загружаемых компонентов, поскольку React.lazy() и Suspense недоступны на стороне сервера. Loadable позволяет рендерить результаты динамического импорта в виде обычных компонентов:

Пример:

import loadable from '@loadable/component'

const OtherComponent = loadable(() => import('./OtherComponent'))

function MyComponent() {

return (

<div>

<OtherComponent />

</div>

)

}

После этого OtherComponent будет загружаться как отдельный "бандл".

### Что такое компонент Suspense?

Если модуль содержит динамический импорт, который не успел загрузиться к моменту рендеринга родительского компонента, необходимо отобразить некоторый запасной контент, например, в виде индикатора загрузки. Это можно реализовать с помощью компонента Suspense.

Пример использования названного компонента:

const OtherComponent = React.lazy(() => import('./OtherComponent'));

function MyComponent() {

return (

<div>

<Suspense fallback={<div>Загрузка...</div>}>

<OtherComponent />

</Suspense>

</div>

);

}

Suspense оборачивает "ленивый" (отложеннозагружаемый) компонент.

### Что такое основанное на роутинге разделение кода?

Одним из лучших мест для разделения кода являются маршруты (routes). Текущая страница полностью обновляется, так что пользователь не сможет взаимодействовать с элементами предыдущей и текущей страницы одновременно. Таким образом, пользовательский опыт не будет испорчен.

Пример разделения кода с помощью библиотеки react-router и функции React.lazy():

import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';

import React, { Suspense, lazy } from 'react';

const Home = lazy(() => import('./routes/Home'));

const About = lazy(() => import('./routes/About'));

const App = () => (

<Router>

<Suspense fallback={<div>Загрузка...</div>}>

<Switch>

<Route exact path="/" component={Home}/>

<Route path="/about" component={About}/>

</Switch>

</Suspense>

</Router>

);

В приведенном примере разделение кода происходит на уровне маршрутов.

### Приведите пример использования контекста

Context - спроектирован для распределения данных, которые можно назвать глобальными, между несколькими React-компонентами, независимо от уровня вложенности этих компонентов.

Пример прямого доступа к пропу theme для стилизации компонента кнопки:

// создаем контекст со значением "light" по умолчанию

const ThemeContext = React.createContext('light');

// создаем компонент App, передающий значение темы всем потомкам

class App extends React.Component {

render() {

return (

<ThemeContext.Provider value="dark">

<Toolbar />

</ThemeContext.Provider>

);

}

}

// промежуточным компонентам не требуется передавать проп `theme` дальше (ниже)

function Toolbar(props) {

return (

<div>

<ThemedButton />

</div>

);

}

// получаем значение темы в компоненте кнопки

class ThemedButton extends React.Component {

static contextType = ThemeContext;

render() {

return <Button theme={this.context} />;

}

}

В функциональном компоненте значение контекста можно получить с помощью хука useContext().

### Для чего используется "дефолтное" значение контекста?

Аргумент defaultValue используется в случаях, когда в дереве компонента не найден подходящий провайдер (provider). Это может быть полезным для тестирования компонентов в изоляции без необходимости их оборачивания в Provider:

const MyContext = React.createContext(defaultValue);

### Как использовать contextType?

contextType используется для потребления (consume) объекта контекста. Данное свойство может использоваться двумя способами:

* 1. **Свойство класса:**

Свойству contextType класса можно присвоить объект контекста, созданный с помощью метода React.createContext(). После этого можно потреблять ближайшее значение контекста посредством this.context в любом методе жизненного цикла, включая метод render():

class MyClass extends React.Component {

componentDidMount() {

let value = this.context;

/\* выполняем побочные эффекты при монтировании, используя значение MyContext \*/

}

componentDidUpdate() {

let value = this.context;

/\* ... \*/

}

componentWillUnmount() {

let value = this.context;

/\* ... \*/

}

render() {

let value = this.context;

/\* выполняем рендерит, используя значение MyContext \*/

}

}

MyClass.contextType = MyContext;

* 1. **Статическое поле:**

contextType можно инициализировать c помощью синтаксиса статических полей класса:

class MyClass extends React.Component {

static contextType = MyContext;

render() {

let value = this.context;

/\* выполняем рендерит, используя значение MyContext \*/

}

}

### Что такое потребитель (Consumer)?

Потребитель - это компонент, подписанный (реагирующий) на изменения контекста. Ему требуется функция в качестве дочернего элемента, получающая текущее значение контекста и возвращающая узел React. Аргумент value равняется пропу value ближайшего провайдера для данного контекста:

<MyContext.Consumer>

{value => /\* выполняем рендеринг на основе значения контекста \*/}

</MyContext.Consumer>

### Как решать проблемы производительности при использовании контекста?

Контекст использует идентификацию ссылок для определения необходимости в повторном рендеринге. Существуют некоторые ошибки, которые могу привести к непреднамеренному рендерингу в потребителях при повторном рендеринге родительского компонента.

Например, в представленном ниже примере все потребители будут перерисовываться при каждом рендеринге провайдера, поскольку каждый раз создается новый объект-значение пропа value:

class App extends React.Component {

render() {

return (

<Provider value={{something: 'нечто'}}>

<Toolbar />

</Provider>

);

}

}

Данная проблема может быть решена путем сохранения состояния в конструкторе родительского компонента:

class App extends React.Component {

constructor(props) {

super(props);

this.state = {

value: { something: 'нечто' },

};

}

render() {

return (

<Provider value={this.state.value}>

<Toolbar />

</Provider>

);

}

}

### Для чего используются forwardRefs в HOC?

Ссылки не могут передаваться потомкам, поскольку они не являются пропами. Они обрабатываются React особым образом, также как ключи. Если вы добавили ссылку к HOC, она будет указывать на самый внешний компонент, а не на оборачиваемый. В этом случае вы можете использовать технику под названием "перенаправление (передача) ссылок".

С помощью API React.forwardRef мы можем передать ссылку внутреннему компоненту:

```javascript

function logProps(Component) {

class LogProps extends React.Component {

componentDidUpdate(prevProps) {

console.log('старые пропы:', prevProps);

console.log('новые пропы:', this.props);

}

render() {

const {forwardedRef, ...rest} = this.props;

// присваиваем кастомный проп "forwardedRef" в качестве ссылки

return <Component ref={forwardedRef} {...rest} />;

}

}

return React.forwardRef((props, ref) => {

return <LogProps {...props} forwardedRef={ref} />;

});

}

```

Используем данный HOC для вывода в консоль всех пропов компонента FancyButton:

```javascript

class FancyButton extends React.Component {

focus() {

// ...

}

// ...

}

export default logProps(FancyButton);

```

Теперь создадим ссылку и передадим ее компоненту. Это позволить установить фокус на кнопку:

```javascript

import FancyButton from './FancyButton';

const ref = React.createRef();

ref.current.focus();

<FancyButton

label="Click Me"

handleClick={handleClick}

ref={ref}

/>;

```

### Почему следует проявлять осторожность при использовании forwardRefs в библиотеке компонентов?

Когда вы начинаете использовать forwardRef в библиотеке компонентов, это, чаще всего, означает несовместимые изменения и релиз новой мажорной версии. Это объясняется изменением поведения библиотеки за счет присвоения ссылок и экспортируемых типов. Такие изменения могут сломать приложение и другие библиотеки.

### Как создать классовый компонент без использования синтаксиса ES6?

Это можно сделать с помощью модуля create-react-class. Для пропов по умолчанию необходимо определить функцию getDefaultProps() в передаваемом объекте. Также требуется реализовать отдельный метод getInitialState(), возвращающий начальное значение:

var Greeting = createReactClass({

getDefaultProps: function() {

return {

name: 'Иван'

};

},

getInitialState: function() {

return {message: this.props.message};

},

handleClick: function() {

console.log(this.state.message);

},

render: function() {

return <h1>Привет, {this.props.name}</h1>;

}

});

**Обратите внимание:** при использовании createReactClass() для всех методов доступно автоматическое связывание, т.е. вам не нужно использовать bind(this) в конструкторе для обработчиков событий.

### Можно ли использовать React без JSX?

Да, JSX не является обязательным условием использования React. На самом деле, JSX используется для того, чтобы избежать настройки компиляции в среде разработки. Каждый JSX-элемент - всего лишь синтаксический сахар для React.createElement(component, props, ...children).

Пример с JSX:

class Greeting extends React.Component {

render() {

return <div>Привет, {this.props.message}!</div>;

}

}

ReactDOM.render(

<Greeting message="World" />,

document.getElementById('root')

);

Тоже самое без JSX:

class Greeting extends React.Component {

render() {

return React.createElement('div', null, `Привет, ${this.props.message}!`);

}

}

ReactDOM.render(

React.createElement(Greeting, {message: 'народ'}, null),

document.getElementById('root')

);

### Что такое алгоритм определения различий?

React нуждается в использовании алгоритма определения эффективного обновления UI для совпадения с последним деревом. Алгоритм определения различий требует выполнения минимального количества операций для преобразования одного дерева в другое. Тем не менее, сложность данного алгоритма составляет порядка O(n3), где n - это количество элементов в дереве.

В этом случае для отображения 1000 элементов потребуется около миллиарда сравнений. Это очень много. Вместо этого, React реализует эвристический алгоритм со сложностью O(n), основываясь на двух предположениях:

* 1. Два элемента разных типов приводят к возникновению разных деревьев
  2. Разработчик может пометить стабильные элементы с помощью пропа key

### Каким правилам следует алгоритм определения различий?

При сравнении двух деревьев, React начинает с сравнения двух корневых элементов каждого поддерева. Поведение различается в зависимости от типов этих элементов. Алгоритм согласования следует таким правилам:

* 1. **Элементы разных типов:** Если корневые элементы имеют разные типы, React уничтожает старое дерево и строит новое с нуля. Например, изменение элемента с <a> на <img> или с <Article> на <Comment> разных типов приводит к полной перестройке
  2. **DOM-элементы одного типа:** При сравнении двух DOM-элементов одинакового типа, React "смотрит" на атрибуты обоих, сохраняет нижележащие DOM-узлы, и обновляет только изменившиеся атрибуты. Вот пример изменения значения атрибута className:

<div className="show" title="React" />

<div className="hide" title="React" />

* 1. **Компоненты одного типа:** При обновлении компонента, экземпляр остается прежним, поэтому состояние сохраняется между рендерами. React обновляет пропы нижележащего экземпляра компонента для совпадения с новым элементом и вызывает методы componentWillReceiveProps() и componentWillUpdate() экземпляра. После этого вызывается метод render() и алгоритм определения различий рекурсивно сравнивает предыдущий результат с новым
  2. **Рекурсивное сравнение потомков:** При рекурсивном сравнении потомков узла DOM, React просто одновременно перебирает оба списка и отмечает различия. Например, добавление нового элемента в конец списка обрабатывается быстро, а в середину медленно, поскольку все элементы после нового будут помечены как новые и перерисованы

<ul>

<li>first</li>

<li>second</li>

</ul>

<ul>

<li>first</li>

<li>second</li>

<li>third</li>

</ul>

* 1. **Обработка ключей:** React поддерживает атрибут key. Когда потомки имеют ключи, React использует их для сравнения потомков оригинального дерева с потомками нового дерева. Поэтому использование ключей существенно повышает эффективность согласования деревьев:

<ul>

<li key="2015">John</li>

<li key="2016">Jane</li>

</ul>

<ul>

<li key="2014">Bob</li>

<li key="2015">John</li>

<li key="2016">Jane</li>

</ul>

### Когда может потребоваться использовать ссылки?

Ссылки используются в следующих случаях:

* 1. Управление фокусом, выделением текста или воспроизведением медиа
  2. Запуск императивной анимации
  3. Интеграция со сторонними библиотеками DOM

### Обязательно ли проп должен называться "render" при использовании рендер-пропов?

Несмотря на название паттерна, использовать "render" в качестве названия пропа необязательно. Любой проп, который является функцией, указывающей компоненту, что следует рендерить, технически является "рендер-пропом".

Пример использования children как пропа для рендера:

<Mouse children={mouse => (

<p>Позиция курсора мыши: {mouse.x}, {mouse.y}</p>

)}/>

На самом деле, проп children не нуждается в имени в списке "атрибутов" JSX-элемента. Вместо этого, его можно поместить снаружи элемента:

<Mouse>

{mouse => (

<p>Позиция курсора мыши: {mouse.x}, {mouse.y}</p>

)}

</Mouse>

При использовании данной техники (без названия), не забудьте явно указать propTypes, что состояние этого потомка должно быть функцией:

Mouse.propTypes = {

children: PropTypes.func.isRequired

};

### В чем заключается проблема использования рендер-пропов в "чистых" компонентах?

Создание функции внутри метода render() противоречит назначению "чистого" компонента. Поскольку поверхностное сравнение пропов будет всегда возвращать false для новых пропов, каждый рендер будут генерировать новое значение для рендер-пропа. Эту проблему можно решить путем определения функции рендеринга в качестве метода экземпляра.

### Как создать HOC с помощью рендер-пропов?

Вы можете реализовать компонент высшего порядка с помощью обычного компонента с рендер-пропом. Например, если вы хотите получить HOC withMouse вместо компонента Mouse, вы можете создать его с помощью Mouse с пропом для рендерига:

function withMouse(Component) {

return class extends React.Component {

render() {

return (

<Mouse render={mouse => (

<Component {...this.props} mouse={mouse} />

)}/>

);

}

}

}

В этом случае рендер-пропы получают возможность использовать другие паттерны.

### Что такое Windowing?

Windowing - это техника, позволяющая рендерить небольшой набор строк в определенный момент, что может существенно уменьшить время повторного рендеринга компонентов, а также число создаваемых DOM-узлов. Если ваше приложение рендерит длинный список данных, тогда рекомендуется использовать эту технику. Популярными решениями в этой сфере являются react-window и react-virtualized. Они предоставляют несколько переиспользуемых компонентов для работы со списками, "гридами" (grids) и табличными данными.

### Как отображать ложные значения в JSX?

Ложные значения, такие как false, null, undefined, как и true являются валидными потомками, но они ничего не рендерят. Для их отображения требуется предварительная конвертация в строку.

Пример:

<div>

Моя JavaScript переменная - это {String(myVariable)}.

</div>

### Назовите типичные случаи использования порталов

Порталы в React используются в случае переполнения (overflow) родительского компонента: скрытые элементы или элементы, которые "вырваны" из контекста стека (стили z-index, position, opacity и т.д.). Данная техника используется для визуализации "независимости" таких элементов от родительских контейнеров.

Примерами подобных элементов могут служить диалоговые или модульные окна, глобальные уведомления, всплывающие подсказки и т.д.

### Как установить значение по умолчанию для неуправляемого компонента?

В React значение атрибута элемента формы перезаписывает соответствующее значение в DOM. В случае использования неуправляемых компонентов вам может потребоваться установить начальное значение, но оставить его неуправляемым при последующих обновлениях. Для решения этой задачи используется атрибут defaultValue вместо value:

render() {

return (

<form onSubmit={this.handleSubmit}>

<label>

User Name:

<input

defaultValue="Иван"

type="text"

ref={this.input} />

</label>

<input type="submit" value="Отправить" />

</form>

);

}

Тоже самое справедливо в отношении полей select и textarea. Однако, для полей checkbox и radio следует использовать атрибут defaultChecked.

### Назовите ваш любимый стек для разработки приложений на React

Несмотря на то, что технический стек различается от разработчика к разработчику, существуют некоторые популярные решения, которые используются повсеместно. Они включают в себя redux, redux-thunk и redux-saga для управления состоянием и работы с асинхронным кодом, react-router для маршрутизации, styled-components для стилизации, axios или fetch для работы с REST API и другие инструменты, такие как Webpack, Babel, reselect и т.д. Если вас интересует данная тема, взгляните на этот проект: <https://github.com/react-boilerplate/react-boilerplate>.

### В чем разница между настоящим и виртуальным DOM?

Ниже представлены основные отличие между реальным и виртуальным DOM:

| Реальный DOM | Виртуальный DOM |
| --- | --- |
| Обновления медленные | Обновления быстрые |
| Манипуляции с DOM очень дорогостоящие | Манипуляции с DOM не очень дорогие |
| Вы можете обновлять HTML напрямую | Вы не можете обновлять HTML напрямую |
| Активная работа с DOM часто приводит к утечкам памяти | Утечки памяти практически полностью исключены |
| При обновлении элемента создается новая DOM | При изменении элемента обновляется только JSX |

### Как добавить Bootstrap в React-приложение?

Bootstrap может быть добавлен в React-приложение тремя способами:

* 1. С помощью Bootstrap CDN: Это простейший способ. Просто добавляем соответствующие теги (стили и скрипт) в head
  2. Bootstrap как зависимость: Если вы используете инструмент для сборки, такой как Webpack, тогда предпочтительной является установка Bootstrap в качестве зависимости:

yarn add bootstrap

// или

npm i bootstrap

* 1. Пакет React Bootstrap: В этом случае можно использовать "компонентную" версию Bootstrap. Для этого существует два популярных решения:
     1. react-bootstrap
     2. reactstrap

### Можете ли вы назвать популярные сайты или приложения, использующие React в качестве фреймворка для фронтенда?

Ниже представлен топ-10 сайтов, использующих React как библиотеку для фронтенда:

* 1. Facebook
  2. Uber
  3. Instagram
  4. WhatsApp
  5. Khan Academy
  6. Airbnb
  7. Dropbox
  8. Flipboard
  9. Netflix
  10. PayPal

### Рекомендуется ли использовать технику "CSS в JS" в React?

React не предоставляет инструмента для работы со стилями. Если вы новичок, тогда хорошей отправной точкой может быть определение стилей в отдельном CSS-файле и ссылка на него через атрибуты className. Функционал по использованию техники CSS-в-JS предоставляется сторонними библиотеками, такими как styled-components.

### Нужно ли мне переписывать все классовые компоненты с помощью хуков?

Нет, не нужно. Вы можете попробовать хуки, переписав с их помощью несколько существующих компонентов (или создав новые), без переписывания всего существующего кода. Команда React не планирует удалять классы (прекращать их поддержку).

### Как запрашивать данные с помощью хуков?

Хук, отвечающий за побочные эффекты, называется useEffect(). Именно этот хук используется для запроса данных с помощью библиотеки axios или встроенного fetch и записи полученных данных в локальное состояние компонента с помощью функции обновления состояния (сеттера), возвращаемой хуком useState().

Пример получения списка статей:

import React, { useState, useEffect } from 'react';

import axios from 'axios';

function App() {

const [data, setData] = useState({ hits: [] });

useEffect(async () => {

const result = await axios(

'http://hn.algolia.com/api/v1/search?query=react',

);

setData(result.data);

}, []);

return (

<ul>

{data.hits.map(item => (

<li key={item.objectID}>

<a href={item.url}>{item.title}</a>

</li>

))}

</ul>

);

}

export default App;

**Обратите внимание:** мы указываем пустой массив в качестве второго аргумента хука во избежание его активации при обновлении компонента. Поэтому хук запускает эффект только один раз во время монтирования компонента.

### Охватывают ли хуки все случаи использования классовых компонентов?

Хуки пока охватывают не все случаи использования классовых компонентов, однако такие планы имеются. В настоящее время не существует хуков, эквивалентых методам жизненного цикла getSnapshotBeforeUpdate() и componentDidCatch().

### Почему мы используем деструктуризацию массива в useState()?

При определении переменной состояния с помощью хука useState(), возвращается массив с двумя элементами. Первым элементом является текущее значение состояния, а вторым - функция для обновления этого значения. Использование [0] и [1] для доступа к указанным элементам массива нецелесообразно, поскольку они имеют вполне определенный смысл. Поэтому мы используем деструктуризацию массива.

Пример получения элемента массива по индексу:

const userStateVariable = useState('userProfile'); // возвращается пара элементов

const user = userStateVariable[0]; // получаем первый элемент

const setUser = userStateVariable[1]; // получаем второй элемент

Пример получения элементов массива с помощью деструктуризации:

const [user, setUser] = useState('userProfile');

### Что послужило источниками для разработки хуков?

Вот некоторые из таких источников:

* 1. Эксперименты с функциональным API в репозитории react-future
  2. Эксперименты сообщества с API рендер-пропов, такие как компонент Reactions
  3. Переменные и ячейки состояния в DisplayScript
  4. Подписки в Rx
  5. Компоненты-редукторы в ReasonReact

### Как получить доступ к императивному интерфейсу веб-компонентов?

Веб-компоненты часто предоставляют императивный API для реализации их функциональности. Для прямого взаимодействия с узлами DOM можно использовать "рефы". Однако, при использовании сторонних библиотек веб-компонентов, лучшим решением является создание React-компонента, оборачивающего веб-компонент.

### Что такое Formik?

Formik - это небольшая библиотека для React, помогающая решать три главные проблемы:

* 1. Запись значений и их получение из состояния формы
  2. Валидация и сообщения об ошибках
  3. Обработка отправки формы

### Назовите популярные решения для работы с асинхронным кодом в Redux?

Среди популярных решений (промежуточных программных обеспечений, middlewares) для работы с асинхронным кодом в экосистеме Redux можно назвать Redux Thunk, Redux Promise, Redux Saga и др.

### Понимают ли браузеры JSX-код?

Нет, браузеры его не понимают. Для преобразования JSX в обычный JavaScript нужен транспилятор. Чаще всего в этой роли выступает Babel.

### Опишите поток данных в React?

React реализует однонаправленный реактивный поток данных с помощью пропов, что ускоряет рендеринг и легче в изучении, нежели традиционное двустороннее связывание данных.

### Что такое react-scripts?

react-scripts - это набор скриптов из инструмента командной строки (CLI) create-react-app, который позволяет быстро начать разработку React-приложения без необходимости предварительной настройки вспомогательных инструментов. Команда react-scripts start, например, запускает сервер для разработки с возможностью "горячей" перезагрузки модулей.

### Назовите основные возможности create-react-app?

Ниже представлен список некоторых возможностей, предоставляемых CRA:

* 1. Поддержка синтаксиса ES6, React, JSX, TypeScript и Flow
  2. Автоматическое добавление CSS-префиксов
  3. Сброс/нормализация стилей
  4. "Живой" сервер для разработки
  5. Быстрый интерактивный запуск юнит-тестов со встроенной поддержкой отчетов о покрытии кода тестами
  6. Встроенный скрипт для сборки JS, CSS, изображений и других статических ресурсов для продакшна с хэшами и картами ресурсов
  7. Сервис-воркер и файл манифеста (manifest), полностью отвечающие критериям прогрессивных веб-приложений

### Для чего используется метод renderToNodeStream()?

ReactDOMServer.renderToNodeStream() используется для генерации HTML на сервере и отправки разметки в ответ на запрос для ускорения загрузки страницы. Он также способствует улучшению SEO, поскольку поисковики могут индексировать только статические страницы.

**Обратите внимание:** данный метод доступен только на сервере.

### Что такое MobX?

MobX - это простое, легко масштабируемое и "проверенное в боях" решение для управления состоянием, позволяющее использовать реактивное функциональное программирование (TFRP). Для React-приложений требуется установка двух библиотек:

yarn add mobx

yarn add mobx-react

// или

npm i mobx

npm i mobx-react

### В чем разница между Redux и MobX?

Основные различия между Redux и MobX состоят в следующем:

| Критерий | Redux | MobX |
| --- | --- | --- |
| Определение | Это JavaScript-библиотека для управления состоянием приложения | Это библиотека для реактивного управления состоянием приложения |
| Программирование | В основном написана с помощью синтаксиса ES6 | Написана на JavaScript (ES5) |
| Хранилище данных | Единственное большое хранилище для всех данных | Возможно использование более одного хранилища |
| Использование | В основном, используется для больших и сложных приложений | Используется для небольших и средних приложений |
| Производительность | Нуждается в улучшении | Обеспечивает лучшую производительность |
| Хранение данных | Используются обычные объекты | Используются наблюдаемые (observable) объекты |

### Обязательно ли изучать ES6 перед изучением React?

Нет, это не обязательно. Однако, многие ресурсы, посвященные React, активно используют синтаксис ES6. Вот парочка примеров:

* 1. **Деструктуризация:** для получения пропов и их использования в компоненте

// ES5

var someData = this.props.someData

var dispatch = this.props.dispatch

// ES6

const { someData, dispatch } = this.props

* 1. **Spread-оператор**: помогает передавать пропы компоненту

// ES5

<SomeComponent someData={this.props.someData} dispatch={this.props.dispatch} />

// ES6

<SomeComponent {...this.props} />

* 1. **Стрелочные функции**: делают синтаксис более компактным

// ES5

var users = usersList.map(function (user) {

return <li>{user.name}</li>

})

// ES6

const users = usersList.map(user => <li>{user.name}</li>);

### Что такое конкурентный рендеринг (Concurrent Rendering)?

Конкурентный рендеринг делает React-приложения более отзывчивыми за счет рендеринга дерева компонентов без блокировки основного пользовательского интерфейса. Он позволяет React откладывать длительный рендеринг для обработки приоритетных событий. При включении данного режима React начинает следить за появлением задач с более высоким приоритетом и при появлении таких задач приостанавливает текущий рендеринг, позволяя этим задачам выполниться в первоочередном порядке. Конкурентный режим можно включить двумя способами:

// 1. Часть приложения

<React.unstable\_ConcurrentMode>

<Something />

</React.unstable\_ConcurrentMode>

// 2. Приложение целиком

ReactDOM.unstable\_createRoot(domNode).render(<App />);

### В чем разница между асинхронным и конкурентным режимами?

По сути, они делают одно и тоже. Раньше "конкурентный" режим назывался "асинхронным". Название было изменено, чтобы подчеркнуть возможность React выполнять работу на разных "приоритетных" уровнях. Также это позволяет не путать его с другими подходами к реализации асинхронного рендеринга.

### Можно ли использовать встроенный JavaScript в React?

Да, вы можете использовать встроенный JavaScript, но будьте готовы получить предупреждение в консоли. Дело в том, что URL, начинающиеся с javascript:, представляют опасность включение необезвреженных строк в такие теги как a, тем самым создавая дыру в безопасности:

const companyProfile = {

website: "javascript: alert('Ваш сайт взломан')",

};

// в консоль будет выведено предупреждение

<a href={companyProfile.website}>Подробнее</a>

В будущем при использовании встроенного JavaScript будет выбрасываться исключение.

### Для чего предназначен плагин ESlint для хуков?

Данный плагин следит за соблюдением правил использования хуков. Он исходит из предположения, что любая функция, начинающаяся с use и следующей за ней большой буквы, является хуком. Эти правила следующие:

* 1. Хуки можно вызывать только внутри компонентов или других хуков (в том числе, пользовательских)
  2. При каждом рендере хуки вызываются в том порядке, в котором они были определены

### В чем разница между императивным и декларативным кодом в React?

Представьте простой UI-компонент, такой как кнопка "Нравится". При нажатии этой кнопки цвет ее фона меняется с серого на голубой, и наоборот.

Императивный способ реализации такой кнопки выглядит следующим образом:

if (user.likes() ) {

if ( hasBlue() ) {

removeBlue();

addGrey();

} else {

removeGrey();

addBlue();

}

}

Требуется проверить, что отображается на экране в данный момент, выполнить все необходимые операции по перерисовке элемента на основе текущего состояния, включая отмену изменений, вызванных предыдущим состоянием. Представьте, каким сложным это может быть в реальном приложении.

Декларативный подход выглядит так:

if ( this.state.liked ) {

return <blueLike />;

} else {

return <greyLike />;

}

Таким образом, декларативный подход разделяет ответственность. Требуется всего лишь определить, как должен выглядеть интерфейс на основе текущего состояния. Кроме того, такой синтаксис гораздо легче читать и поддерживать.

### В чем заключаются преимущества использования TypeScript в React-проектах?

Вот некоторые из преимуществ, предоставляемых использованием TypeScript в React-приложениях:

* 1. Возможность использования самых последних "фич" JavaScript
  2. Возможность использования интерфейсов для определения сложных типов
  3. IDE, такие как VSCode, "заточены" под TypeScript
  4. Возможность устранять ошибки на ранней стадии разработки

### Как обеспечить сохранение информации об аутентификации пользователя между перезагрузками страницы с помощью API управления состоянием контекста (Context API State Management)?

Когда пользователь входит в систему, и затем перезагружает страницу, для сохранения состояния мы обычно используем операцию по загрузке данных пользователя в хуке useEffect() в главном файле (App.js). Это легко реализовать с помощью Redux:

**App.js**

import { loadUser } from '../actions/auth';

store.dispatch(loadUser());

Для того, чтобы получить доступ к контексту в App.js, необходимо обернуть компонент App в компонент AuthState в index.js. После этого при перезагрузке страницы независимо от того, где мы находимся, пользователь останется аутентифицированным (авторизованным), поскольку операция loadUser() будет запускаться при каждом рендеринге:

**index.js**

import React from 'react';

import ReactDOM from 'react-dom';

import App from './App';

import AuthState from './context/auth/AuthState'

ReactDOM.render(

<React.StrictMode>

<AuthState>

<App />

</AuthState>

</React.StrictMode>,

document.getElementById('root')

);

**App.js**

const authContext = useContext(AuthContext);

const { loadUser } = authContext;

useEffect(() => {

loadUser();

},[])

**loadUser**

const loadUser = async () => {

const token = sessionStorage.getItem('token');

if(!token){

dispatch({

type: ERROR

})

}

setAuthToken(token);

try {

const res = await axios('/api/auth');

dispatch({

type: USER\_LOADED,

payload: res.data.data

})

} catch (err) {

console.error(err);

}

}

### В чем заключаются преимущества нового способа преобразования JSX?

Тремя главными преимуществами этого являются:

* 1. Возможность использования JSX без импорта React
  2. Немного уменьшается размер "бандла"
  3. Будущие реализации приведут к уменьшению количества концепций, знание которых требуется для овладения React

### Чем новый способ преобразования JSX отличается от старого?

Новый способ не требует присутствия React в области видимости, т.е. в простых случаях вам не требуется импортировать React.

**Старый способ:**

import React from 'react';

function App() {

return <h1>Доброе утро!</h1>;

}

Данный JSX будет преобразован в JavaScript следующим образом:

import React from 'react';

function App() {

return React.createElement('h1', null, 'Доброе утро!');

}

**Новый способ:**

function App() {

return <h1>Доброе утро!</h1>;

}

Под капотом JSX трансформируется в такой код:

import {jsx as \_jsx} from 'react/jsx-runtime';

function App() {

return \_jsx('h1', { children: 'Доброе утро!' });

}

**Обратите внимание:** вам все еще необходимо импортировать React для обеспечения возможности использования хуков.